Making Use of

Python

Rashi Gupta

Making Use of Python

Making Use of Python

Rashi Gupta

Wiley Publishing, Inc.

Publisher: Robert Ipsen
Editor: Ben Ryan
Managing Editor: Angela Smith
New Media Editor: Brian Snapp
Text Design & Composition: John Wiley Composition Services

This book is printed on acid-free paper. ♾

Published by Wiley Publishing, Inc., New York.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0471-21975-4

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

Introduction

In this competitive age, high productivity, tight deadlines, and short development cycles are the buzzwords in the application development world. These are the reasons why software developers prefer rapid application development (RAD) tools like Python.

Python is a portable, interpreted, object-oriented programming language. It combines remarkable power with very clear syntax. Moreover, its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development.

Python is being used successfully to glue together large software components. It spans multiple platforms, middleware products, and application domains. Python has been around since 1991, and it has a very active user community.

> *Python can fulfill an important integration role in the design of large applications with a long life expectancy. It allows a fast response to changes in user requirements that require adapting the higher-level application logic without changing the fundamental underlying components. It also allows quick adaptation of the application to changes in the underlying components.*
>
> *Guido van Rossum, CNRI*

> *Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.*
>
> *www.python.org*

This book is an attempt to bridge the ever-increasing gap between the market demand and the availability of Python expertise. The first step to becoming an expert is acquiring an in-depth knowledge of Python, and that is exactly what this book has to offer. It begins with the basics of scripting and seamlessly moves to programming intricacies.

Along with conceptual information this book will also provide extensive practical exercises for the reader to gain valuable, real-life exposure to creating different types of applications.

Overview of Python

Python is a free, open-source, general-purpose, interpreted, and powerful scripting language for Web applications. It is an easy yet powerful programming language that provides structure and support for large applications as well as the power and complexity of traditional high-level languages. Python is the ideal choice if you require a single language with the features of both an interpreted and a scripting language.

History of Python

Python is directly derived from the scripting language ABC, which was mainly used for teaching purposes in the 1980s by a small number of people. Python's development was triggered by the need to develop tools to automate monotonous and time-consuming tasks.

Guido van Rossum is the creator of Python. He started work on Python in late 1989 at CWI in Amsterdam. When Guido started work on Python, he was a researcher at CWI. Initially, Python was designed to perform general administration tasks. Later, it became a part of the Amoeba project at CWI and was first released for public use in February 1991. A large part of Python development occurred at CNRI in Reston, Virginia, in the United States. In June 2000, the Python development team moved to Pythonlabs, a member organization of the BeOpen network. The lead developers of Python, including Guido van Rossum, maintained Pythonlabs. In October 2000, the lead developers left BeOpen.com and joined Digital Creations. Since then the team has been involved in Python development. Any intellectual property that is added to Python is taken care by a nonprofit organization called Python Software Foundation.

Features of Python

Python can act as a connecting language that links many separate software components in a simple and flexible manner. It can also act as a guiding language in which high-level Python modules control low-level operations implemented by libraries in other languages. Due to its ease of learning and strength to develop large applications, it can serve both as a learner's first programming language and as an interface for users who want to become experts in advanced application development. Let's discuss some of the salient features of Python.

Easy

Python has an easy syntax, clean and simple semantics, and relatively few keywords, which allow a new developer to learn Python very quickly and easily. It will require a lesser effort for people who have some programming knowledge. Python has a syntax

that is similar to that of Algol, C, and Pascal. In fact, it is a simplification of these languages and does not require any extra effort to learn an unfamiliar concept, syntax, or keywords. Python is an object-oriented programming (OOP) language, but unlike C++, OOP is not a mandatory concept for Python. You can start learning Python and learn about OOP at a convenient point.

Moreover, Python does not have extra symbols for starting and ending code blocks, defining an end to a statement, and pattern matching. Symbols such as curly braces ({}), dollar signs ($), semicolons (;), tildes (~), and at symbols (@), which are part and parcel of many programming languages, do not constrain code written in Python. Indentation is used to group statements to form code blocks. Therefore, you are less likely to have bugs in your code due to incorrect indentations. Python is so simple to understand that a reader who has never seen a single line of code can understand a basic code written in Python.

Scalable

Unix shell scripting languages are fairly easy and can handle simple tasks very easily and efficiently. When you add more features to a script, however, the script becomes very large, complicated, and slow. You are unable to reuse your code, and even small projects require huge scripts. Python provides a better structure and support for large programs than shell scripting. You can build on your code from one project to another or plug or create new components by reusing the existing code. The term “scalability” in relation to Python refers to Python’s capability to provide ingredients to build an application and to provide pluggable and modular architecture for the applications that need to incorporate more functionality.

Python allows you to split your script into modules and reuse these modules in other Python programs. Many standard modules, which can be used based on the requirements of the program, are also built into Python. Many built-in modules aid you in input/output, system calls, socket programming, and GUI programming, such as Tkinter.

High Level

Consider that you have a shell script and you want to add a feature to it. It is possible that the feature involves a system call, variable-length strings, or other data types that are easy to implement in shell but will involve long code passages in C. Perhaps you are not adequately familiar with C to write complex code. Python takes care of all these issues. Python has built-in modules that help you make system calls. Useful, high-level data types, such as lists (resizable arrays) and dictionaries (hash tables) are built into Python, allowing you to express complicated expressions in a single statement. No variable or argument declaration is necessary. After a value is assigned to a name, Python instantly assumes the required type. All this minimizes the time and effort required to implement a particular functionality in a program. The data types also reduce the code size, resulting in a more comprehensible code. On the other hand, these data types would be difficult to implement in C due to the required use of data structures and pointers and the repetitive code needed to implement every large application.

Object Oriented

As stated earlier, OOP is a concept that is not imposed in Python right from the beginning. Nonetheless, Python is a truly object-oriented language and provides features of other structured and procedural languages. All components in Python are objects. Python allows object orientation with multiple inheritance and late binding. You can create object-oriented class hierarchies, and every attribute is referred to in name. attribute notation. In this notation, an attribute is determined dynamically at run time. Python also supports polymorphism—that is, Python callable objects can accept optional arguments, keyword arguments, or an unlimited number of arguments. The same operator can have different meanings according to the elements being referenced. These features allow complex operations to be implemented in small Python declarations. The source code of Python is also object oriented.

Interpreted

Python is an interpreted language that supports byte compilation. Python programs can be run, debugged, and tested interactively by the Python interpreter, which runs in interactive mode. In traditional interpreted languages, execution does not take place in the native binary language of a system. Therefore, execution in traditional interpreted languages is slower compared to that of compiled languages. Python's source code is byte-compiled directly when it is loaded on the interpreter, or it can be explicitly byte-compiled. In addition, byte code of Python is machine independent and can be executed on different hardware and software platforms without compiling it again. Therefore, Python is an intermediate form providing features of both compiled and interpreted languages.

Let's discuss in detail why Python is considered an interpreted language. Python programs can be executed at the interpreter in command-line mode and script mode. In command-line mode, you type Python statements, and the interpreter prints the result.

```
$ python
Python 2.2a4 (#2, Nov 2 2001, 11:00:25)
GCC 2.96 20000731 (Red Hat Linux 7.1 2.96-81)] on Linux2
Type "help","copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

The first line of this example is the command that starts the Python interpreter. The next two lines are messages that are displayed by the interpreter. The third line begins with three greater-than signs (">>>"), which is the prompt used by the interpreter to indicate that it is ready. This is also the interpreter's primary prompt. Typing 1 + 1 at the interpreter returns 2 as the result in the next line.

If your code has a multiline construct, the interpreter prompts with a secondary prompt, which is three dots ("...") by default. Consider the following example:

```
>>> i=1
>>> if i is 1:
```

```
...     print 1+1


2
```

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file written in Python is called a script. For example, a text editor can be used to write a file, calculate.py, with the following contents:

```
print 1 + 1
```

By convention, files that contain Python programs have names with the .py extension. To execute the program, you need to specify the name of the script at the interpreter.

```
$ python calculate.py
2
```

Extensible and Flexible

An application that contains a large amount of code can be effectively organized into smaller modules due to Python's dual structured and object-oriented programming environments. These modules can still interact with each other or with other built-in modules. Python's syntax is the same for accessing both the user-defined and the built-in standard modules. Python is also extremely flexible in the treatment of language components. For example, a Python module that is meant to interact with the external environment can be tested using an imitation of the external environment written in Python.

Rich Core Library

Many development modules are built into Python and are part of the Python Standard Library. A programmer can make use of these tools in the Python Standard Library, depending on the application for which the tools are required. Besides modules that work on all platforms, the library has modules that are specific to a particular platform or environment. Python standard modules perform all types of usual tasks, such as HTTP, FTP, POP, SMTP, and many other services. Using the rich core library, you can write applications for downloading a Web page, parsing HTML files, developing a graphic user interface (GUI), and so on.

Memory Management

C and C++ programmers always need to write code for handling memory management and memory modification even if the program has very little to do with memory access. This always results in an extra burden on the programmer. One clear example is the need for tracking each object and deleting the reserved memory once the object ends its life. This is the responsibility of the developer, and any failure can lead to

memory leaks and other negative consequences. In Python, the interpreter manages memory, thus removing the extra burden on the programmer. This results in fewer errors and a more efficient application involving less development time.

Web Scripting Support and Data Handling

Python is popularly used for developing Internet and intranet applications. Python is well suited for Internet and intranet applications because these applications are highly dynamic and complex, and at times, they need to interact with several environments. Python's dynamism, the ease with which you can write complex applications, and its advanced features, such as HTML, XML, and SGML parsing, allow you to write CGI scripts for several environments.

Object Distribution

You can use Python to implement routines that can communicate with objects in other languages. For example, Python can be used to pass data to COM components. In addition, Common Object Request Broker Architecture (CORBA) can be implemented in Python as well, which enables you to use cross-platform distributed objects.

Databases

Python provides interfaces to all major commercial databases. Besides that, it has built-in modules that enable you to handle flat file databases. It also has object persistence systems that can write entire objects to files. Python's most important database-programming feature, though, is Python API. This API includes functions that make it easy to write applications that communicate with different databases.

GUI Programming

Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix. This is possible using Python's default graphic user interface library, Tkinter. Tkinter is the standard object-oriented interface of the TK GUI API, which is the official GUI development platform of Python.

Extendable and Embeddable

You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient. Compiled extension modules can be created in Python to connect Python modules with external programming libraries or new data types. Extension modules are easily created and maintained using Python. Most platforms support the loading of new compiled components into the interpreter dynamically. Python extension modules can be written in C and C++ for CPython and in Java for JPython.

The Python interpreter can be embedded into another programming or scripting application. Any other program can use the Python interpreter as a simple application program interface (API). Therefore, the Python interpreter can act as a scripting tool that glues everything.

Exception Handling

When running a program, if Python exits due to an error, it generates a complete stack trace of errors. This stack trace indicates the name and type of the error that occurred. The exception handling in Python allows you to detect errors at run time without adding error-checking statements to your code. Exception handlers can be written in Python to defuse a problem, perform a clean-up or maintenance action, or redirect the program flow. This allows a programmer to put in considerably less effort to debug an error.

Portable

Python can run on a wide variety of hardware platforms and has the same interface on all platforms. Its design is not linked to a particular operating system because it is written in portable ANSI C. This means that you can write, test, and upload a program written in Python on Windows, Linux, and Macintosh environments. This depends on whether the application is developed to implement specific commands of an operating system or if the application uses modules that do not work on all platforms. In these situations, the application cannot run on all hardware platforms; however, this affects only a few modules. Usually, the applications that you create run on all the platforms without changing any code.

Freeware

Python is freeware and can be redistributed freely in the source form. The copyright of Python does not allow authors to place it at risk legally and does prevent users from hijacking its copyright. Programmers and users are allowed to use Python's source in any desired way. Programmers can create applications and release them in the binary-only form, which has modules in only the byte-compiled form. The result of the product can, however, be sold or distributed in any manner.

Users and Application Areas of Python

Python is an advanced scripting language that is being used in various areas. Some of the areas where Python is being used are the following:

- Gluing together large software components. These large software components can be written in C, C++, or Java.
- Creating prototypes of an application. The prototype can be written in Python without writing any C, C++, or Java code. Often, the prototype is sufficiently functional and performs well enough to be delivered as the final product, saving considerable development time.

- Writing CGI scripts on all platforms (Unix, Windows, and Mac). Because of this ability Python has a strong presence on the Web.

Besides these, Python is also well represented in the distributed systems world. It is one of the main languages supported by Xerox PARC's ILU. It has also been used to implement the Web browser Grail.

A list of Python users worldwide include the following:

- NASA
- Infoseek
- Digital Creations
- Grail
- Xerox
- Hewlett-Packard
- CMU
- Digital Media Inc.
- University of Queensland, Australia
- Space Telescope Science Institute
- Mind Spring
- Mitretek Systems

Python versus Other Languages

It is a well-known fact that scripting languages are slower than compiled languages. The Python interpreter carries out most of the tasks that are carried out by a compiler in all compiled languages. at Python, however, is an intermediate language that provides the features of both compiled and interpreted languages. Python can be compared with many other languages mainly because it provides many salient features in other languages and is derived from many languages, such as C, C++, Modula-3, ABC, SmallTalk, and Unix shell.

Python is often compared with C and C++ because it has syntax similar to the syntax of these languages. Python is considered a good tool to test C and C++ applications. It also glues some components of C/C++ contributing to C/C++ projects. In many ways, Python has merits over C/C++. Memory allocation and reference errors that occur in C/C++ are eliminated by the Python interpreter, which performs automatic memory management. Python code is usually easier and smaller than that in C and C++. Python's array constructs generate fewer problems than the array constructs of C and C++.

Perl is another scripting language that you can compare with Python. Like Python, Perl is of great use to programmers and system administrators. Perl is also a powerful language for text manipulation and data extraction. Unlike Python, though, Perl has a difficult syntax that dissuades beginners from learning it. Perl is a popular language

used to develop Common Gateway Interface (CGI) scripts for Internet programming. Programmers working on the same large project find it difficult to understand each other's code because there are many ways of writing a program.

Tcl is also one of the popular scripting languages. Python is compared with Tcl for many reasons. Tcl is a powerful and easy scripting language that provides the features of a programming language as well as tools for system calls. Tcl is a more restrictive language than Python because it has fewer data types than Python. Python uses the same toolkit, Tk, as Tcl for developing GUI applications.

Python uses the OOP concept and has syntax similar to that of Java. Unlike Python, Java applications require huge code and a compilation phase. Moreover, Python offers dynamic typing and a rapid development environment. Python, though, is slower and less portable than Java. A breakthrough in the relationship of Python and Java is JPython, a Python interpreter that is constructed completely in Java. It can run on any machine containing Java Virtual Machine (JVM). It provides programmers with the features of Python along with a hoard of Java classes. A complete discussion on JPython is out of the scope of this book. Some of its salient features are as follows:

- JPython provides a scripting environment for Java development.
- JPython generates a truly object-oriented programming environment.
- An application written in JPython can access Java classes directly and can integrate them with its own JPython classes, whenever required.
- JPython provides access to Java AWT/Swing libraries for GUI development.
- Compiled JPython programs create Java byte code, creating a `.class` file, which can be used to create applets.

How This Book Is Organized

This book shrugs away from the traditional content-based approach and uses the problem-based approach to present the concepts of Python. Problems used in the book are presented against the backdrop of real-life scenarios. The problem is followed by a task list that helps to solve that problem, in the process delivering the concepts and their implementation. This practical approach will help readers to understand the real-life application of the language and its use in various scenarios. Moreover, to provide an appropriate learning experience, the concepts will be supported adequately by case studies that will be formulated in such a way that they provide a frame of reference for the reader.

Chapter 1 is a guide to obtaining the Python software and its documentation. It also discusses installation of Python on Unix, Linux, and Windows systems. Finally. it discusses the execution modes of Python and starting Python in Unix, Linux, and Windows.

Chapter 2 is a getting-started guide. It leads into developing a simple Python program. Then, it discusses the standard data types, type operators, and expressions. Finally, it mentions the identifiers and keywords in Python.

Chapter 3 introduces intrinsic operations and input/output. It discusses formatting the output to enhance its visual appeal. It further discusses the built-in functions to use with each data type.

Chapter 4 introduces programming constructs. It discusses using conditional constructs `if...else`, `elif`, and nested `if` constructs. It moves on to discuss loop constructs `while` and `for`. Finally, it discusses `break`, `continue`, and `pass` statements, which are used in loop constructs.

Chapter 5 moves a step further and discusses about functions. It also discusses user-defined functions, Then, it talks about passing functions as arguments and returning values from functions. Finally, it discusses the built-in functions `apply()`, `filter()`, and `map()`.

Chapter 6 discusses organizing code in Python modules. It also delves into importing data from modules into the programming environment. Finally, it discusses organizing modules into packages.

Chapter 7 introduces using files in Python. It discusses writing and appending data to a file. It also discusses how to use Python to read the contents of a file.

Chapter 8 delves into the all-important concept of object-oriented programming. It discusses classes and class objects in Python. Then, it discusses implementing classes. Finally, it talks about using inheritance, overriding methods, and using wrapping.

Chapter 9 explains exceptions and the phases in which the actions related to an exception are performed. Next, it mentions the standard exceptions in Python. It further explains how exceptions can be raised. Finally, the chapter closes by explaining user-defined exceptions.

Chapter 10 moves a few steps further and introduces CGI programming. This chapter assumes that the reader has understands basic Internet concepts and knows how to create Web pages and forms using HTML. For those who are new to the Internet, the chapter briefly recaps World Wide Web, HTTP requests, and HTML form elements and tags. The chapter then differentiates between client-side and server-side scripting. It finally discusses the `cgi` module and generating dynamic Web pages by using a CGI application.

Chapter 11 assumes the reader has basic knowledge about databases, data storage in databases, RDBMS concepts, and their implementation in MySQL. For those who are new to MySQL, this chapter details concepts about installing MySQL and working with the databases and tables in MySQL. It also discusses the Python Database API. Next, the chapter explains the processes of accessing and manipulating a MySQL database by using Python commands. Finally, the chapter discusses concepts such as the creation of a database table to store information and the use of query statements to access and manipulate data.

Chapter 12 delves into network programming in Python by using sockets. It discusses client/server architecture, protocols, sockets, IP addresses, and ports. It then discusses using the socket to create a TCP server, TCP client, UDP server, and UDP client.

Chapter 13 introduces another extremely important concept of multithreaded programming. The chapter begins by differentiating between a single-threaded application and a multithreaded application. It then discusses the `thread` module to create threads. Finally it discusses the `threading` module to create multithreaded applications.

Chapter 14 further discusses advanced Web programming concepts. To start with, this chapter discusses how to create a Web server. Next, it talks about how to work

with URLs by using Python. Finally, this chapter explains advanced CGI to generate dynamic Web pages using cookies and uploading files across an HTTP connection.

Chapter 15 delves into developing user-friendly graphic interfaces. This chapter discusses using Tkinter, the official GUI framework for Python, to create GUI applications. It mentions the various controls that can be included in a GUI interface. Finally, it leads to designing a GUI application.

Finally, the appendix gives a brief introduction to Component Object Model (COM). It mentions the basics of COM and the support for COM in Python.

Who Should Read This Book

This book will be a guide for readers with a basic knowledge of programming. For those with an intermediate knowledge of Python, the book covers the advanced concepts of Python, too. This book will be of great help to people with the following job titles:

- Software engineers
- Web application developers
- Information application developers

The book will provide the necessary skills to create GUI, networking, and Web applications. It will also talk about extending and embedding Python applications.

Tools You Will Need

For performing the tasks in this book, you will need a Pentium 200 MHz computer with a minimum of 64MB RAM (128MB RAM recommended).

You will also need the following software:

- Operating system: Linux 7.1 or Windows 2000 Server
- Web server: Apache 1.3.19-5 (on Linux) and IIS 5.0 (on Windows)
- Relational database management system (RDBMS): MySQL 3.23.36-1
- GNU C++ for Windows 2000
- Python 2.2

What's on the Web Site

The following will be available on the site `www.wiley.com/compbooks/makinguse`:

- Python 2.2
- All the code snippets used in the book

Scenario

All problem statements in this book are based on the scenario of the Techsity University. The following section elaborates on the setup of Techsity University and the university's future plans.

Techsity University

The term *instructor-led training* (ILT) implies that the real strength of the training depends on the instructor and the type of concept insight, knowledge, flexibility, and leadership an instructor can provide through the training. ILT is a form of traditional classroom learning methodology where students can ask questions, seek clarifications, and work directly in coordination with a knowledgeable instructor so as to fully understand concepts and terminology. This was the idea that led to the inception of Techsity University in January 1999.

Techsity University started its operations with 50 students and 4 trainers at its center located in New York City. A total commitment to quality in terms of student satisfaction enabled Techsity University to earn a profit of $1 million in the very first year of its operation. As an outcome of student responses to the courses offered, student enrollment and staff recruitment in Techsity University increased over the past three years.

Currently, Techsity University provides 50 instructor-led courses, which include soft skills development courses and technical courses. Currently, the university offers these courses in five cities in different states of the United States; however, only four courses are available at any given time. At present, Techsity University offers regular as well as part-time courses.

The five cities in which the Techsity University has centers are these:

- New York
- Los Angeles
- Chicago
- Denver
- Washington, D.C.

The courses offered by Techsity University can be classified in the following categories:

- Business development
- Professional development
- Information technology
- Software
- Desktop technologies

Course Structure

Usually, the duration of the courses offered by the Techsity University varies from three to four weeks. Details about each course, such as fees, the syllabus, and the class structure, are available at the front office of each University location. Typically, a course comprises a beginner, an intermediate, and an advanced level. A student may choose to start from any of the three levels. If a student chooses to start a course from the intermediate or advanced levels, the student is interviewed at the beginning of the course. In addition, the student needs to take an entry-level test so that the authorities can determine whether the student meets the course prerequisites. Therefore, a student can join a course a level or two above the beginner's level only after clearing the test and the interview.

The schedule of regular courses consists of a five-hour class from Monday to Friday. Not all courses are offered as part time. The part-time courses have five-hour classes on weekends and two-hour classes on two chosen days of the week.

Course objectives, syllabi, and any preliminary reading assignments are given to students before the start of the course. Depending on the type of course, the course structure comprises theoretical classes and hands-on practice classes. To increase the effectiveness of courses, a class may also contain an amalgamation of both theory and practice.

Fee Structure

The University has a flexible and moderate fee structure for the convenience of its students. A student can choose to pay the entire fee at the time of enrollment or pay the fee amount in installments. When paying in installments, a student has to pay 50 percent of the fees as down payment, 25 percent after the completion of 25 percent of the course, and the remaining 25 percent after completion of 75 percent the course.

Future Direction

As the number of students approaching the University for enrollment is increasing, it is considering offering its courses on the Web as well. The main reasons for such a consideration are as follows:

- Accommodating ILT in a student's schedule means rearranging the student's life around training rather than arranging training around the student's life.
- Because ILT relies so severely on the instructors, a bad instructor can negate all the advantages associated with ILT.
- For the employed, attending ILT means time away from the office and involves additional costs for travel, lodging, meals and so forth.
- ILT is conducted at a speed dictated by the training material, the time allotted to the class, and the instructor's approach to the training material. Students who do not fit the knowledge base or the understanding of the intended target audience in the class may find an ILT class a frustrating experience. A good instructor will tune the presentation to make it applicable to the widest range of classroom audience.

Keeping these points in mind, Techsity University plans to gradually launch its content on the Web. The Techsity University Web site planned to be developed soon will not only offer its Web-based courses but also promote instructor-led training, which forms the backbone of the courses it offers. The Web site will also provide support to students, such as providing experts to answer students' queries and accepting and evaluating student assignments online.

CHAPTER

1

An Introduction to Python

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Identify the scenarios where Python can be used**
- **Obtain Python and its documentation**
- **Examine system requirements**
- **Install Python**
- **Start Python in different execution modes**

Getting Started

The Introduction familiarized you with Python. It discussed what Python is, its history, and its key features. This chapter will discuss the locations from where Python can be obtained. You will also learn how to install Python on the various platforms and the various ways in which a Python program can be executed. At the end, you will learn about the Web sites from which Python documentation can be obtained.

Understanding Requirements

Problem Statement

Techsity University has been set up recently and currently does not have an online site. The University plans to expand its activities in the field of online training courses so that students from all over the United States can enroll for courses. Techsity wants a Web site that will enable learners to obtain information about its courses at the touch of a key. The Web site should be fast to code, scalable, and robust.

Techsity has an application for internal use that was developed using C and C++. It wants to reuse as much of this application as possible to reduce development time for the new application. The new application needs to be developed within three months by using the existing team of C and C++ developers.

Techsity's Chief Technology Officer (CTO) has recommended Python as the new language because it meets all the requirements of the new application and because the existing team will be able to learn the new language easily, thus reducing the learning and development time. The CTO has assigned the task of understanding system requirements, obtaining Python for the development team, and getting Python running to Jim.

Task List

- **Determine requirements of the University.**
- **Download Python and its documentation.**
- **Determine the system requirements.**
- **Install Python.**
- **Start Python in different execution modes.**

Determine Requirements of the University

Before deciding on the software application and hardware platforms to use for the previous scenario, let's understand the requirements of the University (see Table 1.1).

Obtain Python and Its Documentation

Python is currently available in five stable versions. Python 1.5.2 was released in April 1999. Python 1.6 was made available to the public in September 2000 and has major new features and enhancements over Python 1.5.2. Python 2.0, released in October 2000, was more of a transition from Python 1.6. Python 2.1.1, released in July 2001, was mainly a bug fix release for Python 2.1. The final release of the latest version, Python 2.2, was released in December 2001. This book was developed when Python 2.2 was in its alpha release 4. Therefore, most of the screen shots in this book are taken in Python 2.2a4.

Table 1.1 Requirements of Techsity University

REQUIREMENT	DESCRIPTION
Development time	The entire application needs to be developed in three months.
Platform	The customers and dealers should be able to use any kind of operating system platform; that is, the application should be platform independent.
Speed	Techsity wants a computerized system that enables learners and dealers to obtain information about its products and schemes readily.
Accessibility	The University wants an information system that will enable individuals from any part of the United States to receive help on the courses and schemes offered by Techsity.
Association	The system should allow extension and embedding of C and C++ to make use of existing applications in these languages.
Other features	The CTO wants the application to be powerful, robust, and scalable.

How to decide on the version? The code written in Python 1.5.2 is backward compatible with older versions and is available on the largest number of platforms. Python 2.0 has new features, such as Unicode support, but does not support backward compatibility. Programmers who are migrating from Python 1.5.2 to 2.0 can use Python 1.6. Programmers looking for improved features, such as Python's model of objects and classes, improved multiple inheritance, new iteration interface, and new and improved modules, should use Python 2.2.

You can find all the latest information about Python on the Python official Web site or the Pythonlabs Web site. The links are as follows:

```
http://www.python.org                        (Community home page)
http://www.pythonlabs.com                    (Commercial home page)
```

Python is freeware; therefore, all of Python's current software versions are available for free on the sites listed previously. You can download the Python distribution for Unix, Windows, and Mac systems from the link www.python.org/download. In addition, Python documentation, news, and more are also available on this site. You can download the Python documentation from www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats. A part of the documentation is

also available with the software package. The Python 2.2 distribution is a part of the following documentation that helps you learn Python and its advanced features:

- The Python Tutorial
- Global Module Index
- Library Reference
- Macintosh Module Reference
- Installing Python Modules
- Language Reference
- Extending and Embedding
- Python/C API
- Documenting Python
- Distributing Python Modules

NOTE If you do not have Internet connectivity, all the versions of Python are available on the Web site for this book, www.wiley.com/compbooks/gupta.

Determine the System Requirements

As discussed earlier, Python is available on a wide variety of platforms, such as Unix, Windows, Macintosh, X Windows, OS/2, Be-OS, VMS, and Amiga. Python is supported by most of the platforms that have a C compiler. After you download your version of Python, which is in compressed format, you need to unpack the downloaded files. If you are using Unix, the GNA gzip program performs the required action. The GNA gzip program is available at www.gnu.org/software/gzip/gzip.html. For Windows, it is necessary to have the WinZip program to unpack the downloaded files. Winzip can be downloaded from www.winzip.com. To run Python, the system requirements are as shown in Table 1.2.

You can choose from a host of software platforms to run Python. For the development of this book, the software configuration shown in Table 1.3 is used.

Table 1.2 Hardware Specifications for Using Python

HARDWARE	SPECIFICATION
Processor	Pentium, 200 MHz
RAM	64 MB, 128 MB (Recommended)

Table 1.3 Software Specifications for Using Python

SOFTWARE	SPECIFICATION
Operating system	Linux 7.1, Windows 2000 Server, and Windows NT Server
Web server	Apache 1.3.19-5, (IIS 5.0 for Windows)
RDBMS	My SQL 3.23.36-1
Web browser	Netscape 4.76
GNU C++	For Windows NT/2000
Python	Version 2.2a4

Install Python

As mentioned earlier, Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python. If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation. For example, for Windows, you might want to install the additional Tcl/Tk support feature on which you can build your GUI applications or install Win32 extensions, COM extensions, and more. Similarly, you can choose to install Tkinter or the GNU readline module, which allows you to scroll back through Python commands in the interpreter. All this can be incorporated in the Unix installation of Python by editing the Setup file in the Modules subdirectory in the source distribution.

Unix Installation

After unpacking the files in the source distribution and converting them into a `tar` archive, you can install Python. This requires running the `./configure` script in the Python archive. Then, type `make` at the shell prompt, and type `make install`. On Unix machines, the Python executable is usually installed in the `/usr/local/bin` directory, and its libraries are installed in `/usr/local/lib/pythonXX` where `XX` is the version of Python that you are using.

Linux Installation

Almost all major Linux distributions include Python. You might want to install the latest version of Python, though, even if Python is installed automatically for you. This can be done by using the RPM (RedHat Packet Manager) package for installation. The

RPMs for Linux installation can be downloaded from www.python.org/2.2/rpms.html. You must execute the following command to update the RPM:

```
rpm -Uhv python2.2-2.2b1-2.i386.rpm
```

Or use the following command to install the RPM from the RPM package:

```
rpm -ihv python2.2-2.2b1-2.i386.rpm
```

(Note that the preceding filenames reflect the beta version of Python 2.2.) For a fresh installation of Python from the source code on a Linux machine, follow the same steps as for Unix installation. You can also download the source RPM and build a binary package by using the following command:

```
rpm —rebuild python2.2-2.2b1-2.src.rpm
```

Windows Installation

On Windows, you can install Python by running Python-XXX.exe, where XXX is Python's latest release. On Windows, double-clicking the file will launch the Installation Wizard, as shown in Figure 1.1.

After installing Python, if you want to install PythonWin and PythonCOM software also, double-click the `win32all-YYY.exe` file. Each version of Python has a specific corresponding `win32all` file. Therefore, do not install the file that is intended for a different release. You can download this file for the specific version of Python from the following location: http://aspn.activestate.com/ASPN/Downloads/ActivePython/Extensions/Win32all.

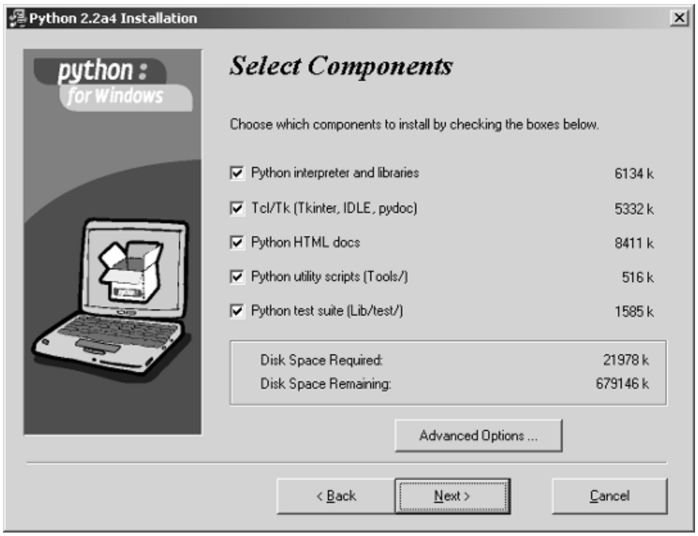


Figure 1.1 Python Installation Wizard guides you through a simple installation process.

Start Python in Different Execution Modes

You can start Python in three different ways. One way is to start the interpreter interactively where each line that you enter is executed at the same time. The second way is to run a script written in Python. In this case, the interpreter directly executes the script. Finally, you can run the interpreter in the form of a GUI that is part of the Integrated Development Environment (IDE). An IDE usually provides tools for debugging and editing text.

Interactive Interpreter

You can start Python in the command-line interpreter mode and start writing code. Any operating system that provides you with a command-line interpreter or a shell window, such as DOS or Unix, can start Python in an interactive interpreter mode. This mode can be extremely helpful when you want to test the specific features of Python.

Unix. To start the Python interpreter, you need to type the full path to the Python executable if you have not added the directory containing the Python executable to your search path. Python usually exists in `/usr/bin` or `/usr/local/bin` directories. To add Python to the search path, you can add the full path of the directory containing the Python executable to the set path or PATH= directive. After this, refresh the shell's path variable. Now, you can invoke the Python interpreter by typing `python` at the shell prompt.

```
$python
```

Typing `python` at the shell prompt will start the Python interpreter in the Unix environment and will show the Python primary prompt denoted by ">>>" as shown in Figure 1.2.

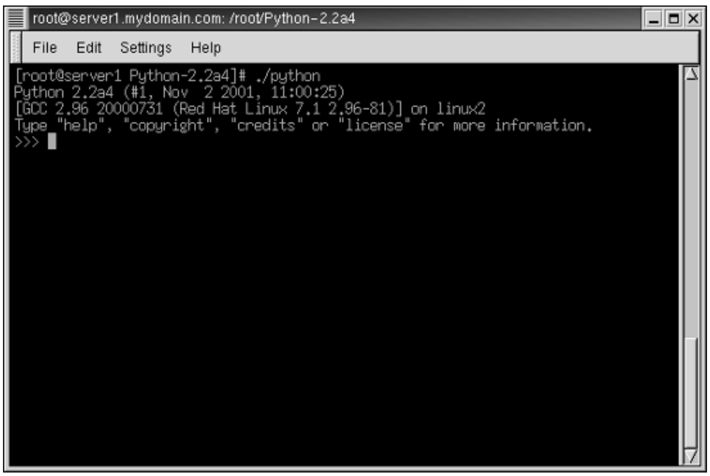


Figure 1.2 Starting Python in a Unix window.

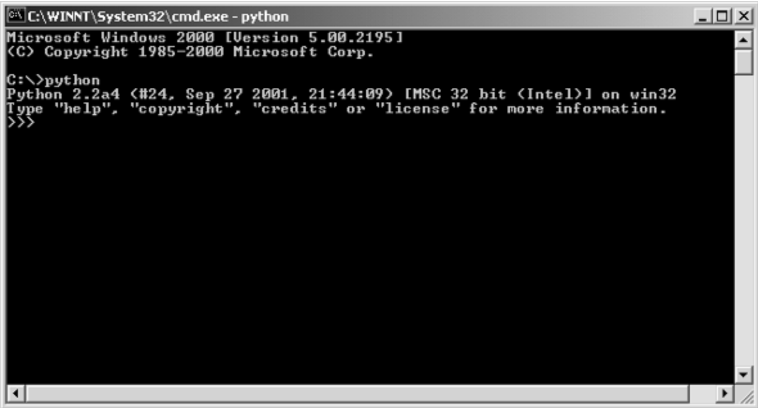


Figure 1.3 Starting Python in the Windows command line.

Windows. In order to run Python directly by typing `python` at the command prompt, add Python to your search path. This is similar to what is done in Unix. You can do this by editing the `C:\Autoexec.bat` file. The Python executable in Windows is usually installed in `C:\Program Files\Python` or `C:\Python`. In Windows also, you can start Python by typing `python` at the command prompt `C:\`. This will start the Python interpreter in Windows, as shown in Figure 1.3.

Script from the Command Line

You can also request Python to directly execute a script from the command-line interface. This is the same for Windows, Unix, or any other operating system that supports command-line interface, as in the following commands:

```
C:\>python myscript.py
$ python myscript.py
```

These commands, for Windows and Unix, execute the script `myscript.py` from the current directory. If the script you want to execute is not in the current directory, specify the complete path for the script.

You can also invoke the Python interpreter automatically without explicitly invoking it from the command line. Include the following line to launch shell as the first line of your script:

```
#!/usr/local/bin/python
```

The path following "#!" is the full path of the location of the Python interpreter. Be careful to give the correct path name; if the path name is not correct, the shell will return an error message.

When you have added a startup directive to the beginning of your script, the Python interpreter does not need to be explicitly invoked. You can run the script directly using:

```
$myscript.py
```

Alternatively, you can use a command named `env` for the startup directive, which is installed in either `/bin` or `/usr/bin`. This command finds the Python interpreter in your path. You can use `env` when you do not know where the Python executable is located. You can also use `env` if you change its position frequently, but it is still available through the directory path you specify.

In Windows, if you have a Python IDE installed (this will be discussed in the next section), you can execute a script directly by double-clicking it.

Integrated Development Environment

Python can also be started from a graphical user interface (GUI) environment. This can be done using a GUI application, such as Tcl/Tk. Most GUI applications are IDEs as well. IDEs provide the additional features of editing, tracing errors, and debugging.

Unix. IDLE is the first Unix IDE for Python. IDLE is Tkinter based and requires Tcl/Tk to be installed on your system. You do not need to install Tcl/Tk fully because the current versions of Python include the minimal subset of the Tcl/Tk library in the distribution.

The IDLE executable is located in the Tools subdirectory with the source distribution. IDLE can be invoked by typing `idle` at the shell prompt. Figure 1.4 shows the IDLE window in Unix.

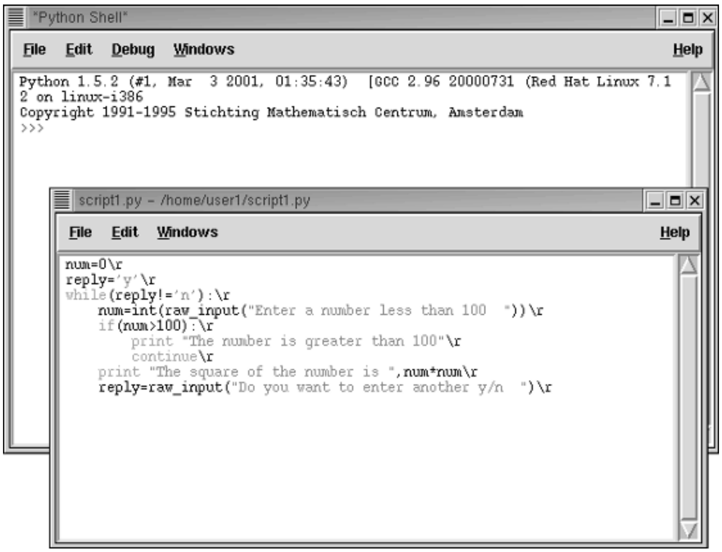


Figure 1.4 Starting IDLE in Unix.

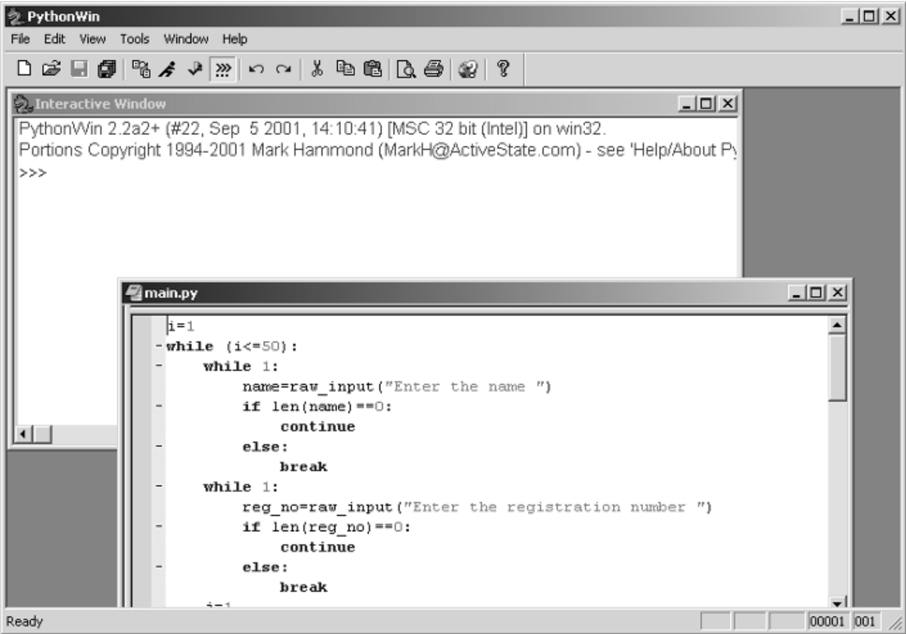


Figure 1.5 The PythonWin environment in Windows.

Windows. PythonWin is the first Windows IDE for Python. The PythonWin distribution includes Win32 API extensions, COM extensions, and Tcl/Tk. PythonWin can be invoked by executing the file `pythonwin.exe`, which is usually located in the same directory as Python in the `Pythonwin` subdirectory `C:\Program Files\Python\Pythonwin`. Among its main features are a color text editor, a debugger, an interactive shell window, and more. Figure 1.5 shows the PythonWin environment running in Windows, including the main PythonWin window and a script open in its integrated source code editor.

As mentioned earlier, PythonWin can be installed by running the executable file `win32all-YYY.exe` specific for the version of Python that you have on your computer. You can obtain more information on PythonWin from the Pythonwin readme file, which is located at `C:\Program Files\Python\Pythonwin\readme.html` or any other location where PythonWin is installed.

Besides PythonWin, IDLE can also be installed on the Windows platform. It is in the `Tools` or `Idle` subdirectory of the folder where Python is installed. To start IDLE, double-click the `idle.pyw` executable. Figure 1.6 shows the IDLE window in Windows.

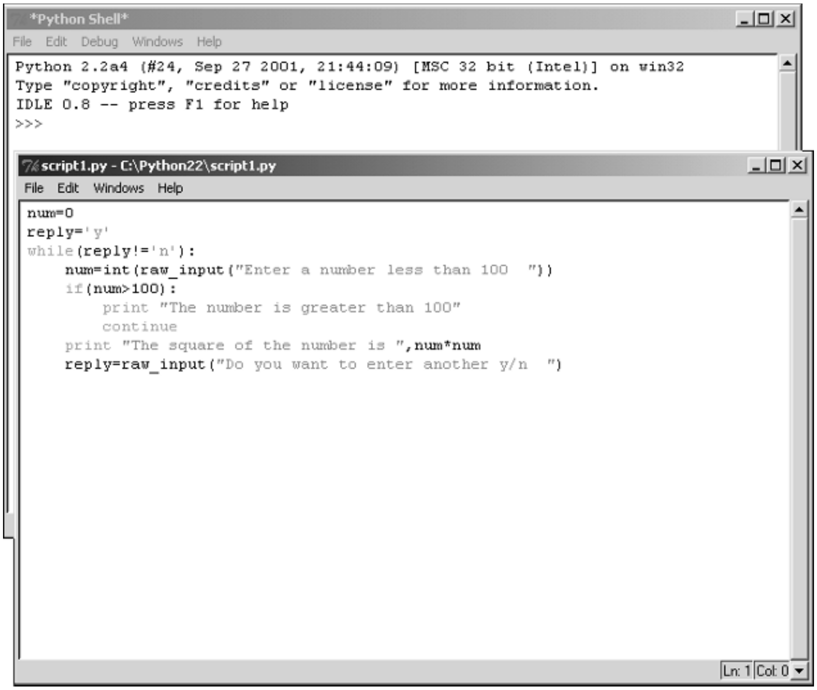


Figure 1.6 The IDLE environment in Windows.

Macintosh. The Python version that runs on Macintosh is called MacPython. It is also available on www.python.org and can be downloaded as MacBinary or BinHex'd files. Python source code is also available on the main Web site as a Stuff-It archive, and the full version is available as a unique file, which also includes Tkinter and IDLE. As in Unix and Windows, IDLE also works on Macintosh. Figure 1.7 shows the IDLE environment in Macintosh.

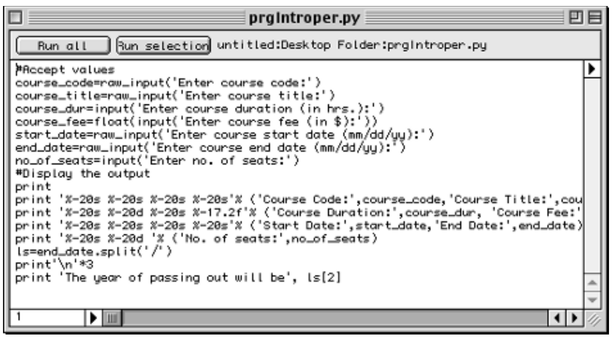


Figure 1.7 The IDLE environment in Macintosh.

Summary

In this chapter, you learned the following:

- Python is currently available in five stable versions: 1.5.2, 1.6, 2.0, 2.1.1, and 2.2. The final release of the latest version Python 2.2 was made available in December 2001.
- You can find all the latest information about Python on the Python official Web site or the Pythonlabs Web site. The links, respectively, are:

  ```
  http://www.python.org              (Community home page)
  http://www.pythonlabs.com          (Commercial home page)
  ```

- Python runs on a wide variety of platforms, such as Unix, Windows, Macintosh, X Windows, OS/2, Be-OS, VMS, and Amiga, to name a few.
- You can download the Python distribution for Unix, Windows, and Mac systems from the link www.python.org/download. In addition, the Python documentation, news, and other articles are also available on this site. You can download the Python documentation from www.python.org/doc/. This documentation is available in HTML, PDF, and PostScript formats.
- To install Python, download the binary applicable for your platform and execute it in the way applicable for your platform.
- Python can be started in three different ways:
  - As the interactive interpreter
  - Directly executing a script from the command line
  - As an Integrated Development Environment (IDE)

CHAPTER

2

Getting Started with Python

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Write your first Python program**
- **Use comments**
- **Use Python as a calculator**
- **Use variables**
- **Use standard data types:**
  - **Number**
  - **String**
  - **List**
  - **Tuple**
  - **Dictionary**
- **Examine memory management in Python**

Getting Started

This chapter will introduce the core part of Python and will familiarize you with Python syntax. In this chapter, you will learn to write your first Python program. Then, it will discuss the standard data types in Python. You will also learn about the standard type operators and how variables and operators can be combined to form expressions. Finally, you will learn about identifiers and keywords in Python.

Let's now learn how to store information in a variable and how to use the stored information in various ways, and discuss the relevance of expressions and operators. This chapter uses the scenario of Techsity University, which needs to store and display student details using different variables. You will also learn to use lists and dictionaries to store all the information about students. The chapter has been designed so that each task identified for the problem statement will progress toward equipping you with all the knowledge you will need to solve the problem statement.

Writing Your First Python Program

The first program that we will write is the Hello World program, which is typically the first program for learning any programming or scripting language. Type the following command at the interpreter.

```
>>>print 'Hello World!'
```

The output of this command will be:

```
Hello World!
```

As you can see in this command, the `print` statement is used to display the output on the screen. Program input and output are the two most important basic features of any programming language. In Python, the program output can be obtained using the `print` statement. In order to enable the program to interact with the user, it needs to accept data input from a user. To obtain user input from the command line, the easiest way is to use the `raw_input()` built-in function. The `raw_input()` function accepts only text input. For example, for accepting user input for a user name and then displaying it, you can use the following commands:

```
>>>name=raw_input('Enter your name: ')
```

This command shows the following line where the user can enter text.

```
Enter your name: Laura
```

Now, to print the value input by the user, use the following `print` statement.

```
>>>print 'Your name is: ', name
```

Figure 2.1 shows how the preceding statements appear on the interpreter.

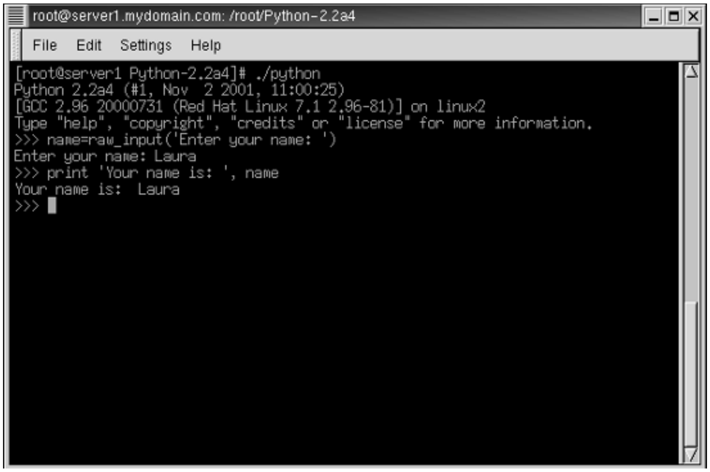


Figure 2.1 A sample program to accept user input and display the output on the screen.

NOTE A *function* is a block of code that is used to perform a single task. *Built-in functions* are defined internally in Python and are ready to be called to perform a specific task. You will learn about functions in detail in Chapter 5, "Functions."

A program usually does much more than just accept data from the user and display it. To enhance the readability of a complex code, lines that explain the code are added to the code. These lines do not affect the program in any way and are called comments. Let's learn more about using comments in Python.

Comments

As in most shell scripting and other scripting languages, you can use the hash or pound (#) sign to start a comment. A comment begins from the hash or pound sign and continues until the end of the line.

```
>>># First comment
...print 'Learn about comments' # second comment'
```

The output of the previous statements will be:

```
Learn about comments
```

Note that the first line of code doesn't execute because it is preceded by a # sign. Comments enhance the understanding of the code both for the programmer and for other people who want to use the code. They should be kept clear, short, and simple.

You should take care that comments serve the purpose they are meant for, and you should avoid them when they are not required.

Python as a Calculator

The previous chapter talked about various execution modes of Python. Chapter 1, "An Introduction to Python," discussed that each line of code can be executed right after typing at the Python interpreter. Therefore, the Python interpreter can be used as a simple calculator. You can write an expression at the interpreter, and it will return the resulting value. Expressions are the same as the ones in most programming languages, such as C, C++, and Pascal. They use the well-known +, -, *, and / operators. Let's consider some examples.

```
>>>2+2
4
```

Note that in the preceding example, the statement `2+2` is at Python's primary prompt. In addition, the output of one line is shown directly in the line below it and is indicated by the absence of >>>.

```
>>>#Learn using Python interpreter as calculator
...2+2
4
```

Note that in the preceding example, the comment is at Python's primary prompt. The statement `2+2` is at Python's secondary prompt, three dots, `...`.

```
>>>(60+40)/10
10
```

The division of two integers using the / operator returns the floor value shown as follows:

```
>>>9/2
4
>>>9/-2
-5
```

This chapter will further describe various operators and how they can be grouped with other Python types in the later sections. After becoming familiar with the Python syntax, you will learn how to use variables in Python.

Using Variables in Python

Programming is all about working with data. While programming, you often access memory directly or indirectly to store or retrieve data. In some programming languages,

such as C and C++, you can access the memory directly; in other programming languages, such as Visual Basic and Java, you cannot access memory directly. One thing common across all programming languages is the use of variables to store data in memory. Therefore, variables play a big role in any form of programming. Before describing variables in detail, let's discuss how Python uses objects to store data.

As we discussed in the previous chapter, Python is an object-oriented programming (OOP) language, but you don't need to use OOP as a concept to create a working application in the beginning. We will discuss OOP in detail in Chapter 8, "Object-Oriented Programming." We have briefly introduced objects here, though. Python uses objects for all types of data storage. Any entity that contains any type of value or data is called an *object*. You can classify all data as an object or a relation to an object. Each object has three characteristics: identity, type, and value.

Identity. The identity of an object refers to its address in the computer's memory. Identity is also a unique identifier that makes it distinct from the rest of the objects. You can use the `id()` built-in function to obtain the memory address of a variable.

Type. The type of an object determines the operations that are supported by an object. It also defines the values that are possible for objects of that type and the operations that can be performed on that object. The `type()` built-in function can be used to determine the type of the Python object.

Value. The value of an object refers to the data item contained in that object.

The identity and type characteristics of an object are read only and are assigned when the object is created. Objects whose values can be changed without changing their identity are called *mutable* objects, and those whose values cannot be changed are called *immutable* objects.

Some Python objects have multiple attributes and can store many data items. In addition, these objects might contain executable code that you can use to perform desired tasks. These built-in object types are files, functions, classes, modules, and methods.

Although objects can store multiple data items, there are certain objects that store a value and have a single attribute. These objects are called *variables*.

Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. Who decides how much memory is to be reserved and what should be stored in this memory? This is done by assigning data types to variables. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables. Consider another situation in which you need to store a large amount of the same type of data. One way to do this is to declare multiple variables and then remember the names of all these variables! A simpler way to do this in Python is by using tuples, lists, or dictionaries.

Assigning Values to Variables

Unlike those of other languages, Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. Like most other programming languages, the equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable, and the operand to the right of the = operator is the value stored in the variable. For example,

```
>>>price=100
>>>discount=25
>>>price-discount
75
```

Here, `100` and `25` are the values assigned to `price` and `discount` variables, respectively. The expression `price-discount` calculates the difference between `price` and `discount`. Similarly, string values can also be assigned to variables. For example,

```
>>>a='play'
>>>b='ground'
>>>a+b
'playground'
```

The concatenation of multiple string values can also be assigned to a variable directly by using the plus (+) operator.

```
>>> c='Py'+'thon'
>>> c
'Python'
```

After you have assigned a value to a variable, you can use that variable in other expressions. For example,

```
>>>a=2
>>>a=a+3
>>>a
5
```

Note that you do not have to explicitly use a `print` statement to display the output in the interpreter. Simply writing the name of the variable at the interpreter will display the value contained in a variable.

Multiple Assignment

You can also assign a single value to several variables simultaneously. For example,

```
>>>a=b=c=1
>>>a
1
```

```
>>>b
1
>>>c
1
```

In the preceding example, an integer object is created with the value `1`, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example,

```
>>>a,b,c=1,2,'learn types'
>>>a
1
>>>b
2
>>>c
'learn types'
```

In the preceding example, two integer objects with values `1` and `2` are assigned to variables `a` and `b`, and one string object with the value `'learn types'` is assigned to the variable `c`. This type of assignment is a special case in which items on both sides of the equal sign are tuples. We will learn about tuples later in this chapter. You can also use parentheses for multiple assignments, but otherwise they are optional.

```
>>>(a,b,c)=(1,2,'learn types')
```

This technique of multiple assignment can be used to swap values of variables. For example,

```
>>> (x,y)=10,20
>>> x
10
>>> y
20
>>> (x,y)=(y,x)
>>> x
20
>>> y
10
```

You have learned that values can be assigned to variables. The question that arises next, however, is about the type of values that can be assigned to variables in Python. Let's learn about Python's standard types to help you answer the question.

Standard Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has some standard types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

These standard types are also referred to as primitive data types or just data types. Out of these, lists, tuples, and dictionaries are compound data types, which are used to group other values.

Using Numbers

Number data types store numeric values. They are immutable data types, which means that changing the value of a number data type results in a newly allocated object. Like all data types, number objects are created when you assign a value to them. For example,

```
>>>var=1
```

An existing number can be changed by assigning a value to it again. The new value can be related to the old value, can be another variable, or can be a completely new value.

```
>>>var=var+1
>>>var=2.76
>>>floatvar=6.4
>>>var=floatvar
```

When you do not want to use a particular number, usually you just stop using it. You can also delete the reference to a number object by using the `del` statement. When this is done, the name of the variable cannot be used unless it is assigned to another value. The syntax of the `del` statement is:

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the `del` statement. For example,

```
>>>del var
>>>del varab,varcd
```

Python further classifies numbers into four types:

Regular or Plain integer. Plain integers are the most common data types among all languages. Most machines allow you to assign a value to an integer variable from -2^{31} to $2^{31} - 1$. A plain integer in Python is implemented as the data type

`signed long int` in C. Integers are usually represented in the base 10 (decimal) format. They can also be represented in the base 8 (octal) and base 16 (hexadecimal) formats. Octal values are prefixed by "0", and hexadecimal values are prefixed by "0x" or "0X". Some examples of plain integers are these:

```
10          100         6542        -784
083        -042        -0X43         0X61
```

Long integer. Long integers are helpful when you want to use a number out of the range of plain integers that is less than -2^{31} or greater than $2^{31} - 1$. There is virtually no limit to their size except that the size is limited to the available virtual memory of your machine. Virtual memory is a constraint because when any variable is assigned or used, it is loaded into memory. Therefore, it is necessary that memory have enough space to load the variable. The suffix "l" or "L" at the end of any integer value denotes a long integer. Like plain integers, their values can also be in decimal, octal, and hexadecimal. Some examples of long integers are these:

```
53924561L       -0x19423L        012L     -4721845294529L
0xDEFABCECBDAECBFBAEl    535133629843L     -052418132735L
```

NOTE Python allows you to use a lowercase L, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Even if you assign a long integer to a variable that has a lowercase L, Python displays long integers with an uppercase L.

```
>>>varlong=812386l
>>>varlong
812386L
```

Floating-point real number. This type of number is also referred to as float. Floating-point real numbers occupy 8 bytes on a 64-bit computer where 52 bits are allocated to the mantissa, 11 bits to the exponent, and the last bit for the sign. This gives you a range from $+10^{308.25}$ through $-10^{308.25}$. The float data type in Python is implemented as the `double` data type in C.

Float values can have two parts, a decimal point part and an optional exponent part. The decimal point part contains a decimal value, and the exponent part contains a lowercase or uppercase "`e`" followed by the appropriate nonzero exponential value. The positive or negative sign between "`e`" and the exponent denotes the sign of the exponent. The sign is also optional, and its absence indicates a positive exponent. Some examples of floating point numbers are these:

```
0.0             14.5        -15.4        32.3+e18
-90.76712      -90.         -32.54e100   70.2-E12
```

Complex number. A complex number consists of an ordered pair of real floating-point numbers denoted by `a + bj`, where `a` is the real part and `b` is the imaginary part of the complex number. The imaginary part of the complex number is

suffixed by a lowercase j (`j`) or an uppercase j (`J`). Some examples of complex numbers are these:

```
3.14j     45.j          .876j          5.43+3.2j          3e+26J
4.53e-7j          9.322e-36j          -.6545+0J
```

The imaginary and the real parts of a complex number object can be extracted using the data attributes of the complex number. In addition, a method attribute of complex numbers can be used to return the complex conjugate of the object. The following examples help you to understand these attributes:

```
>>>complexobj=23.87-1.23j
>>>complexobj.real
23.870000000000001
>>>complexobj.imag
-1.23
>>>complexobj.conjugate()
(23.870000000000001+1.23j)
```

NOTE The concept of conjugates is used in relation with complex numbers. For any complex number of the form a + ib, the conjugate is a - ib and vice versa.

Arithmetic Operators

Operators play an important part in performing calculations. Besides arithmetic operators, Python also supports conditional operators for making value comparisons. We will cover conditional operators in a later chapter. Among the arithmetic operators that Python supports are the unary operators + and - for no change and negation, respectively, and the binary arithmetic operators, `+`, `-`, `*`, `/`, `%`, and `**`, for addition, subtraction, multiplication, division, modulo, and exponentiation, respectively. In an arithmetic expression such as:

```
x = y + z
```

`y` and `z` are called the operands for the + operator. Table 2.1 describes each type of arithmetic operator that can be used to perform calculations in Python.

A new operator added to Python 2.2 is `//`, which is used for floor division. Floor division is the division of operands where the result is the quotient in which the digits after the decimal point are removed. This is different from true division where the result is the actual quotient. The / operator, which is present in all versions of Python, performs floor division for integer values and true division if one or both values are floating-point numbers. For example, `9/2` is equal to `4` and `-9/2` is equal to `-5`. However, `9.0/2` is equal to `4.5` and `-9/2.0` is equal to `-4.5`. The // operator performs floor division for all types of operands. Therefore, `9//2` is equal to `4` and `9.0//2.0` is equal to `4.0`.

Table 2.1 Arithmetic Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
+	Adds the operands.	`x = y + z`	Adds the value of y and z and stores the result in x.
-	Subtracts the right operand from the left operand.	`x = y - z`	Subtracts z from y and stores the result in x.
**	Raises the right operand to the power of the left operand.	`x = y ** z`	y is raised to the power of z and stores the result in x.
*	Multiplies the operands.	`x = y * z`	Multiples the values y and z and stores the result in x.
/	Divides the left operand by the right operand.	`x = y / z`	Divides y by z and stores the result in x. Performs floor division if both operands are plain integers, and performs true division if either or both operands are floating-point numbers.
%	Calculates the remainder of an integer division	`x = y % z`	Divides y by z and stores the remainder in x.

If an expression involves more than one operator, Python uses precedence rules to decide which operator is to be evaluated first. If two operators have the same precedence, Python uses associativity rules for evaluating an expression. For example,

```
>>>x=7+3*6
>>>x
25
>>>y=100/4*5
>>>y
125
```

To understand the preceding output, consider Table 2.2, a precedence and associativity table for arithmetic operators.

Table 2.2 Associativity Table for Arithmetic Operators

TYPE	OPERATORS	ASSOCIATIVITY
Value construction	`()`	Innermost to outermost
Exponentiation	`**`	Highest
Multiplicative	`//  *  /  %`	Left to right
Additive	`+  -`	Left to right

The `**` operator has the highest precedence. The operator `*` has a higher precedence than the operator `+`, and the operator `/` has the same precedence as `*`. In the expression `x = 7 + 3 * 6`, the part `3 * 6` is evaluated first and the result `18` is added to `7`. In the expression `y = 100 / 4 * 5`, the part `100/4` is evaluated first because the operator `/` is to the left of the operator `*`.

You can change the precedence and associativity of the arithmetic operators by using `()`. The `()` operator has the highest precedence among the three types being discussed. The `()` operator has left to right associativity for evaluating the data within it. Therefore, we have:

```
>>> x = (7 + 3) * 6
>>>x
60
>>> y = 100 / (4 * 5)
>>>y
5
>>> z = 7 + ( 5 * (8 / 2) + (4 + 6))
>>>z
37
```

The modulus operator `%` returns the remainder of an integer division. The following example explains the working of the modulus operator:

```
>>> 7 % 3
1
>>>0 % 3
0
>>>1.0 % 3.0
1.0
```

The exponentiation operator behaves noticeably for unary operators. If it has a unary operator to its right, it takes the power to be raised after applying the operator to it. If there is a unary operator with the number to its left, it raises the power first and then applies the operator. For example,

```
>>> 5**2
25
>>> 5**-2
```

```
0.04
>>> -5**2
-25
>>> (-5)**2
25
```

You have learned how values are assigned to variables by using the = operator. Let's see the other assignment operators available in Python.

Assignment Operators

If you want to assign expressions to variables, you can also use the assignment operators listed in Table 2.3.

Table 2.3 Assignment Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`=`	Assigns the value of the right operand to the left.	`x = y`	Assigns the value of `y` to `x`. `y` could be an expression, as shown in the previous table.
`+=`	Adds the operands and assigns the result to the left operand.	`x + = y`	Adds the value of `y` to `x`. The expression could also be written as `x = x + y`.
`-=`	Subtracts the right operand from the left operand and stores the result in the left operand.	`x - = y`	Subtracts `y` from `x`. Equivalent to `x = x - y`.
`*=`	Multiplies the left operand by the right operand and stores the result in the left operand.	`x * = y`	Multiplies the values `x` and `y` and stores the result in `x`. Equivalent to `x = x * y`.
`/=`	Divides the left operand by the right operand and stores the result in the left operand.	`x / = y`	Divides `x` by `y` and stores the result in `x`. Equivalent to `x = x / y`.

Continues

Table 2.3 Assignment Operators (Continued)

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`%=`	Divides the left operand by the right operand and stores the remainder in the left operand.	`x % = y`	Divides `x` by `y` and stores the remainder in `x`. Equivalent to `x = x % y`.

Any of the operators listed in Table 2.3 can be used as shown here:

```
x <operator>= y
```

can also be represented as

```
x = x <operator> y
```

That is, y is evaluated before the operation takes place. For example,

```
>>> x=15
>>> y=12
>>> x+=y
>>> x
27
>>> x+=y*2
>>> x
51
```

Notice that the assignment `x+=y` is treated as `x=x+y`. After performing the first operation, the value of `x` becomes `27`. Then, in the second operation, `12*2=24` is added to `x` to return `51`.

Using Strings

Strings are one of the most commonly used types in Python. A string can be defined by enclosing characters within single or double quotes. For example,

```
>>> str='Hello World!'
>>> str
'Hello World!'
>>> astr="Welcome!"
>>> astr
'Welcome!'
>>> print astr
Welcome!
```

In the preceding example, notice that when the output of a variable containing a string value is directly displayed by typing it at the interpreter, string quotes appear. When the same string is printed using the `print` statement, only the string appears without string quotes.

A string, which is created using single or double quotes, cannot contain the same type of quotes inside it. For example, `str='couldn't'` gives you a syntax error because the interpreter takes the second quote as the end of the string and therefore does not recognize the characters following it. There are different ways of assigning such strings to variables. You can have a single quote(s) inside a string enclosed in double quotes or vice versa. Quotes can be followed by a backslash (\) to make the quotes a part of a string. For example,

```
>>> str="couldn't"
>>> str
"couldn't"
>>> astr='couldn"t'
>>> astr
'couldn"t'
>>> "\"No,\" she said"
'"No," she said'
```

If you have a long string that spans multiple lines, a backslash (\) followed by `n` can be used to break into multiple lines. For example,

```
>>> str="Python is an easy yet powerful programming language, \n which
provides structure and support for large applications \n as well as the
power and complexity of traditional high-level languages."
```

`\n` inserts new lines in the string when you print the string `str` by using the `print` statement. The new lines are inserted as follows:

```
>>> print str
Python is an easy yet powerful programming language,
 which provides structure and support for large applications
 as well as the power and complexity of traditional high-level languages.
```

You can also enclose a string in triple quotes, such as `"""` or `'''`. When you use triple quotes, new lines do not have to be escaped by using special characters, but they will still be a part of the string. For example,

```
>>>str="""Python is an easy yet powerful
programming language,
which provides structure and support for
large applications as well as power
and complexity of traditional high-level languages."""
```

Print the string as follows:

```
>>> print str
Python is an easy yet powerful
programming language,
which provides structure and support for
large applications as well as power
and complexity of traditional high-level languages.
```

You can concatenate a string by using the + operator and replicate a string by using the * operator. For example,

```
>>> strvar='play'+'ing'
>>> strvar
'playing'
>>> newstr=strvar*4
>>> newstr
'playingplayingplayingplaying'
```

You can change the preceding statement to assign a value to `strval` by using the following statement:

```
>>> strval*=4
>>> strval
'playingplayingplayingplaying'
```

In addition, two strings enclosed within quotes written next to each other are automatically concatenated. For example,

```
>>> strvar='play''ing'
>>> strvar
'playing'
```

The length of a string can be found using the `len()` function.

```
>>>len(strvar)
7
```

Unlike many other languages, Python does not support the character type. You can use the string type to extract a single character or a substring from a string. This method of extracting a single character or a substring from a string by using the index or indices is called *slicing*. Slice notation consists of two indices separated by a colon, and it can be used to extract substrings. For example,

```
>>> str='learn'
>>> str[0]
'l'
>>> str[0:2]
'le'
>>> str[0:4]
'lear'
>>> str[0:5]
'learn'
```

Note that in the preceding example, the length of the string `str` is `5`. When counting forward, the indices start from `0` at the left and end at one less than the length of the string. Therefore, for the previous string, any substring can be accessed within the range `0` through `4`.

A string can also be counted backward, starting from the index `-1`, which corresponds to the rightmost character to the negative value of the length of the string, which is the index of the first character in the string. The following representation shows how the indices can be counted forward and backward in a string.

```
 0      1      2      3      4
 l      e      a      r      n
-5     -4     -3     -2     -1
```

For example,

```
>>> str[-1]
'n'
>>> str[-5]
'l'
>>> str[-5:-1]
'lear'
>>> str[1:-1]
'ear'
```

When you miss the starting or ending index, the beginning index defaults to zero and the ending index defaults to the size of the string being sliced.

```
>>> str[:3]
'lea'
>>> str[2:]
'arn'
```

The omission of both indices returns a copy of the string.

```
>>>str[:]
'learn'
```

Python string data type is an immutable data type, and therefore, once created, it cannot be changed. If a value can be assigned to a string variable, however, why is it an immutable data type? The reason is simple. When you assign a different value to the variable containing a string object, a new object is created. Let's see how.

```
str1='know'
>>> str1
'know'
>>> id(str1)
16971488
>>> str1='treat'
>>> id(str1)
17043008
```

Notice that when the string `str1` is created, its identity given by the `id()` function is different from the identity obtained after changing its value. This indicates that a

completely new object is formed. Assigning a value to an index position in the string results in an error.

```
>>> str[0]='u'
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in ?
    str[0]='u'
TypeError: object doesn't support item assignment
```

You can create a new string combining the new content with the old string.

```
>>> 'whir'+str[0]
'whirl'
>>> 'y'+str[1:]
'yearn'
```

We have discussed the number and string data types. Now, let's discuss a scenario in which you can use your learning.

Problem Statement

As a member of a team that is developing the Web site for Techsity University, you have been assigned the task of creating a software module that displays the following student details:

- Name
- Registration number
- Date of birth
- Address
- City
- Home phone number
- Score in subject 1
- Score in subject 2
- Average score

Based on the inputs that you have gained, let's look at the tasks you need to perform to solve the preceding problem.

Task List

✓ **Identify the variables and data types to be used.**

✓ **Write the code to display the details.**

✓ **Execute the code.**

Identify the Variables and Data Types to Be Used

Based on the knowledge acquired and the information given in the problem statement, the variables and their data types listed in Table 2.4 can be identified.

Write the Code to Display the Details

After identifying the variables and their types let's write the code to assign values and display the details of a student.

```
# Program to display user details
reg_no='S001'
name='Michael'
dob='10/8/1971'
add='Amityville'
city='San Francisco'
phno='534-177-3312'
score1=70
score2=85
print "Student details are:"
print '-'*20
print "Registration Number:", reg_no
print "Name:",name
print "Date of Birth:", dob
print "Address:",add
print "City:",city
print "Phone number:", phno
avg_score=(score1+score2)/2
print "Average Score:", avg_score
```

Table 2.4 Variables and Data Types Identified for the Problem Statement

VARIABLE NAME	DATA TYPE	DESCRIPTION
`reg_no`	String	Registration number
`name`	String	Student name
`dob`	String	Date of birth
`add`	String	Address
`city`	String	City
`home_phno`	String	Home phone number
`score1`	Integer	Score in first subject
`score2`	Integer	Score in second subject
`avg_score`	Float	Average score

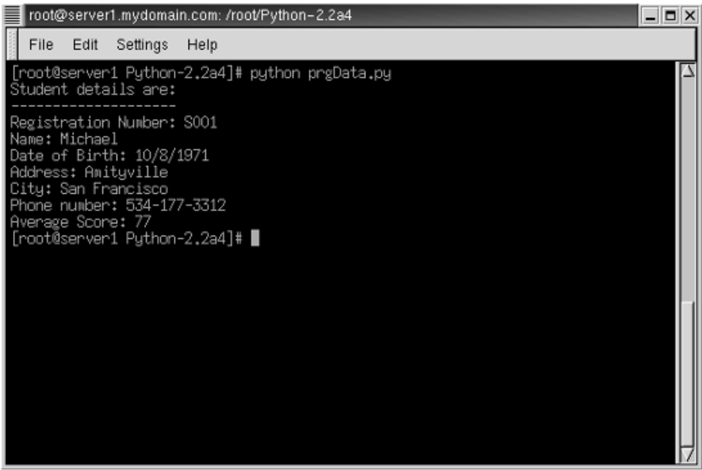


Figure 2.2 The output of the first problem.

Execute the Code

To be able to view the output of the preceding code, the following steps have to be executed:

1. Type the code in a text editor.
2. Save the file as **probstat1.py**.
3. Make the directory where you have saved the file the current directory.
4. On the shell prompt, type:

```
$ python probstat1.py
```

Figure 2.2 shows the sample output.

You now have basic knowledge about the two data types, numbers and strings. Let's move on to the list, which is a sequence data type in Python.

Using Lists

Consider a situation where the information regarding 50 models of a car needs to be stored. An integer or a string is capable of storing only one value at a time. In addition, it is not easy to define and keep track of 50 integers or strings in a program. The solution is to declare one variable with 50 elements to store the information about the various car models. To solve this problem, Python allows the use of three compound data types: lists, tuples, and dictionaries.

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets (`[]`). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. For example,

```
>>> listvar=['abcd',123,2.23,'efgh']
>>> listvar
['abcd', 123, 2.23, 'efgh']
```

Lists can be sliced in the same way as strings, and the same slicing rules for positive and negative indices apply to lists as well. For example,

```
>>> listvar[0]
'abcd'
>>> listvar[2]
2.23
>>> listvar[1:-1]
[123, 2.23]
```

Lists can also be concatenated and replicated as in the following examples:

```
>>> listvar[:3]+['fake',3*2]
['abcd', 123, 2.23, 'fake', 6]
>>> 3*listvar
['abcd', 123, 2.23, 'efgh', 'abcd', 123, 2.23, 'efgh', 'abcd', 123, 2.23,
'efgh']
>>> 2*listvar+['genuine']
['abcd', 123, 2.23, 'efgh', 'abcd', 123, 2.23, 'efgh', 'genuine']
```

We have seen that strings are immutable data types and that individual elements in strings cannot be changed. You can, however, change the individual elements in lists.

```
>>> listvar
['abcd', 123, 2.23, 'efgh']
>>> listvar[3]=8-4j
>>> listvar
['abcd', 123, 2.23, (8-4j)]
```

You can also nest lists; that is, a list can consist of another list. You can perform all the operations of a list on that element. For example,

```
>>> listvar[1]=[888,'pqr']
>>> listvar
['abcd', [888, 'pqr'], 2.23, (8-4j)]
```

You can also extract the slice of the slice. For example,

```
>>> listvar[1][0]
888
```

You can use the `len()` function to find the length of a list. For example,

```
>>> len(listvar)
4
>>> listvar[1]
[888, 'pqr']
```

You can even assign elements to slices. This might involve replacing, removing, or inserting items in a list. Here are a few examples:

```
>>> listvar[1][0]
888
>>> listvar[0:2]=[]
>>> listvar
[2.23, (8-4j)]

listvar[1:1]=[1234,'ddd']
>>> listvar
[2.23, 1234, 'ddd', (8-4j)]

>>> listvar[1:2]=['007','xyz']
>>> listvar
[2.23, '007', 'xyz', 'ddd', (8-4j)]

>>> listvar[:0]=listvar
>>> listvar
[2.23, '007', 'xyz', 'ddd', (8-4j), 2.23, '007', 'xyz', 'ddd', (8-4j)]
```

Items can also be removed from a list by using the `del` statement if you know the index position of the items, which have to be deleted. If you do not know the index position, you can use the remove method.

```
>>> del listvar[0:5]
>>> listvar
[2.23, '007', 'xyz', 'ddd', (8-4j)]
>>> del listvar[0]
>>> listvar.remove('xyz')
>>> listvar
['007', 'ddd', (8-4j)]
```

You can add items at the end of a list by using the append method. For example,

```
>>> listvar.append([123,'abcd'])
>>> listvar
['007', 'ddd', (8-4j), [123, 'abcd']]
>>> listvar[3].append(999)
>>> listvar
['007', 'ddd', (8-4j), [123, 'abcd', 999]]
```

You have learned about the two data types, strings and lists, in which you can extract and manipulate items by indexing and slicing. Due to this reason, Python terms these data types as *sequence* data types. Python also supports two more sequence data types, tuples and dictionaries.

Using Tuples

A tuple is another sequence data type that is similar to the list. When you are writing code, there are situations in which the mutability offered by the list data type can

become a hindrance to the functionality of specific code. Therefore, Python has another data type that is immutable, which is called a tuple. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. For example,

```
>>> tup=(123,'abc',345)
>>> tup
(123, 'abc', 345)
```

You can input a tuple without enclosing it in brackets; however, parentheses are important when tuples are a part of larger expressions. For example, you may want to nest tuples—that is, include a tuple in another tuple.

```
>>> atup='hello',532
>>> atup
('hello', 532)
>>> anothertup=tup,('a','b','c')
>>> anothertup
((123, 'abc', 345), ('a', 'b', 'c'))
```

The main difference between lists and tuples is that tuples are immutable; it's not possible to assign values to or remove individual items in a tuple. Assigning a value to an item or removing it by using its index gives an error.

```
>>> atup[1]=999
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

You can use indices or slices to access values in a tuple. For example,

```
>>> tup[1]
'abc'
>>> tup[0:2]
(123, 'abc')
>>> anothertup[:2]
((123, 'abc', 345), ('a', 'b', 'c'))
>>> anothertup[1][0]
'a'
```

To overcome the problem of immutability in a tuple, you can take portions of existing tuples to create a new tuple. For example,

```
>>> tup=(123,'abc',345)
>>> tup
(123, 'abc', 345)
>>>tup1=('ggg',1562)
>>> tupmain=tup+tup1
>>> tupmain
(123, 'abc', 345, 'ggg', 1562)
```

In the previous example, two tuples, `tup` and `tup1`, are created individually. They are assigned to another tuple, `tupmain`, by using the + operator. Note that the tuple `tupmain` contains the values of both `tup` and `tup1`. This method can also be used to add elements to the same tuple.

```
>>> tup=('a','b')
>>> tup=tup+('c','d')
>>> tup
('a', 'b', 'c', 'd')
```

You can also create tuples with mutable objects, such as lists. Therefore, you can change the values in the list. For example,

```
>>> tuple=(567,'ddd',[123,3214,'abc'])
>>> tuple
(567, 'ddd', [123, 3214, 'abc'])
>>> tuple[2][1]=5678
>>> tuple
(567, 'ddd', [123, 5678, 'abc'])
```

In the previous example, first we created a tuple containing a list. Then, we successfully changed the first item in that list. Therefore, even if tuples are immutable, mutable items contained in them can be changed.

If you want to create an empty tuple, you can assign a pair of parentheses to a variable. The length of an empty tuple is 0. For example,

```
>>> tup=()
>>> tup
()
>>> len(tup)
0
```

If you want to create a tuple containing a single item, then a comma should follow that item. Without a trailing comma, the type of the element is assumed as the type of the variable being assigned instead of a tuple. For example,

```
>>> single=('welcome')
>>> len(single)
7
```

In the preceding example, because you assigned a single item to `single`, the length of the tuple should be `1`, which contains a single string, `'welcome'`. The length is `7` because `single` now contains the string `'welcome'` instead of a tuple. When a single data item is enclosed within parentheses, it acts as a binder instead of as a delimiter for tuples. To assign a variable to a single item, the item should be followed by a comma. For example,

```
>>> tup=('welcome',).
>>> tup
('welcome',)
>>> len(tup)
1
```

Using Dictionaries

The sequence data types that you have learned use a range of values to index the values in them. You have seen how indexing and slicing are used to access data items in strings, lists, and tuples. What is the solution if you want to access a data item by using a *key,* not an index? Dictionaries are useful in such situations. Dictionaries use *keys* to index values in them. A dictionary is analogous to a telephone directory, which is used in daily life. You look for the telephone number of a person based on the name of the person. Similarly, in a dictionary, the values are mapped according to keys. The key does not have to be a numeric value to index the data item; it can be any immutable data type, such as strings, numbers, or tuples. A tuple can be used as a key only if it does not contain any mutable object directly or indirectly. In other words, a Python dictionary is an unordered set of *key:value* pairs. Python dictionaries are similar to Perl's associative arrays or hash tables.

The keys in a dictionary are unique; one key can be associated with only a single value. The syntax of the dictionary entry is key:value. A dictionary is enclosed within curly braces ({}). Each key:value pair is separated by a comma. The output of a dictionary is also shown in the same way.

```
>>> dict={}
>>> dict1={'name':'mac','ecode':6734,'dept':'sales'}
>>> dict1
{'ecode': 6734, 'dept': 'sales', 'name': 'mac'}
>>> dict
{}
```

In the preceding example, we created an empty dictionary `dict` by assigning a pair of curly braces to it and another dictionary `dict1` by assigning three key:value pairs separated by commas to it.

A value can be extracted from a dictionary by using the key associated with it. For example,

```
>>> dict1['dept']
'sales'
>>> dict1['ecode']
6734
```

In the preceding example, the value associated with the key `'dept'` is `'sales`'. When you use a key belonging to a dictionary, the corresponding value is displayed. You use keys for lookup in a dictionary instead of indices. In other words, it is possible to access the data item by using the key associated with it. Any attempt to access a key, which does not exist in the dictionary, gives an error. For example,

```
>>> dict1['telno']
(most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'telno' is not defined
```

The dictionary is a mutable data type. Therefore, key:value pairs can be added to it any number of times. In the other sequence data types that we discussed, both lookup

and assignment are achieved using an index. In a dictionary, a new value can be added or an old value can be changed by supplying a key enclosed in square brackets as an argument. Therefore, for a dictionary, both lookup and assignment are achieved using a key instead of an index.

```
>>> dict1['telno']='555-451243'
>>> dict1
{'dept': 'sales', 'telno': '555-451243', 'name': 'mac', 'ecode': 6734}
```

In the preceding example, because `'telno'` is not an existing key in `dict1`, a new key:value pair will be added to `dict1` with key=`'telno'` and value=`'555-451243'`.

It is not necessary for all the keys in a dictionary to belong to the same data type. Let's look at another dictionary where the keys are of different data types.

```
>>> dict3={'2':1234,2:'abc',6.5:'troy' }
>>> dict3
{6.5: 'troy', 2: 'abc', '2': 1234}
```

In the preceding example, the first key `'2'` is a string and is associated with the integer `1234`; the second key is the integer `2` and is associated with a string `'abc'`; the third key is a float `6.5` and is paired with a string `'troy'`.

If you want to extract all the keys in a dictionary, you can use the `keys()` method of the dictionary object. The `keys()` method returns a list of the keys in random order. For example,

```
>>> dict3.keys()
[6.5, 2, '2']
```

The length of a dictionary is the number of key:value pairs in it.

```
>>>len(dict3)
3
```

Items can be removed from a dictionary by using the `del` statement. You can also use the `del` statement to delete the entire dictionary, but usually you will not require this. You can also clear the dictionary by using the `clear()` method.

```
>>> del dict1['ecode']     #Remove an entry with 'ecode'
>>> dict1
{'name': 'mac', 'telno': '555-451243', 'dept': 'sales'}
>>> dict1.clear()          #Remove all entries from dict1
>>> dict1
{}
>>> del dict1              #Delete dict1
>>> dict1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'dict1' is not defined
```

We learned how you can assign values to variables in Python. Let's see which words can be used to name the variables you define.

Identifiers and Keywords

We have learned that a program refers to a variable by using its name. Every programming language defines a set of rules, which must be respected to build variable names. Such names are called *identifiers*. Among all names allowed in Python, certain identifiers are reserved by the language and cannot be used as programmer-defined identifiers. These names are called *keywords*. Python uses the following identifiers as keywords.

KEYWORDS

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
while	def	finally	in	print
continue	exec	import	pass	
class	except	if	or	yield

The Python interpreter defines certain rules for naming identifiers, which are as follows:

1. Variable names should not have any embedded spaces or symbols such as `? ! @ # + - % ^ & * ( ) [ ] { } . , ; : " ' /` and `\`. Underscores _ can be used wherever space is required. For example, `basic_salary`.
2. Variable names must be unique. For example, to store four different numbers, four unique variable names need to be used. Identifiers are case-sensitive; that is, uppercase letters are considered distinct from lowercase letters.
3. A variable name can have any number of characters.
4. A variable name must begin with a letter or an underscore, which may be followed by a sequence of letters, digits (0 through 9), or underscores. The first character in a variable name cannot be a digit.

NOTE The Python is a case-sensitive language. This means that the variable `customerName` is not the same as the variable `customername`.

Different organizations lay down certain guidelines to be followed by a programmer while naming identifiers. These guidelines are aimed at improving the readability of a program. Your program will compile even if you do not follow these guidelines for naming identifiers. Generally, it is a good practice to follow these variable-naming conventions.

- Variable names must be meaningful and short. The names must reflect the data that the variables contain. For example, to store the age of an employee, the variable name can be `employee_age`.
- Variable names are normally written in lowercase letters.
- If a variable name contains two or more words, join the words and begin each word with an uppercase letter. Otherwise, separate each word with an underscore.

The following variable names are valid:

```
address1
employee_name
this_variable_name_is_very_long
```

The following variable names are invalid:

```
#phone
1stName
```

According to the Python standard, identifiers containing a double underscore (__) or beginning with an underscore (_) are reserved for use by Python implementations and standard libraries and should not be used as ordinary identifiers.

In this chapter, you have already learned that variable types are not declared in Python. When you start writing longer code in later chapters, you will also learn that you can assign a value to a variable whenever you need it; it is not necessary that all variables in a program be declared in the beginning. This is not the case in many other languages, though. Variables have to be declared by specifying the type they belong to in the beginning of a code block. Python does not impose any such restriction; variables are automatically declared during the first assignment. After the variable has been assigned to an object, it can be accessed using its name. Now, let's see how the Python interpreter manages memory.

Memory Management

Python declares the type and memory space required by a variable at run time. When you create a variable, the Python interpreter creates an object whose type is determined by the type of value you assign to it. After the object is created, a reference to that object is assigned to the variable, which is on the left-hand side of the assignment statement.

The interpreter handles the memory management by itself. We have learned that when we assign a value to a variable, a certain amount of space in the memory is allocated to that variable. Therefore, in this process, some resources from the system are borrowed and need to be returned to it eventually. In Python, all this is handled by the interpreter automatically. When a variable is no longer being used—that is, it is not using the memory—it is reclaimed to the system. This mechanism is called *garbage collection*. The Python garbage collector automatically deallocates the objects that are no longer required. Python enables the programmer to concentrate on the application

being written, and the programmer need not worry about the lower level of resource management tasks. Garbage collection is done by tracking the number of references made to an object. This is referred to as *reference counting*.

Reference Counting

Python performs reference counting by keeping track of the number of references made to an object in an internal variable called *reference counter*. When an object is created, the reference counter is set to 1. Each time a reference is made to the variable, the reference count is incremented by 1. This reference is made when another variable is assigned to the same object. The reference can also be made when the variable is passed as an argument to invoke a function, a method, or a class instantiation or assigned as a data item in a sequence. The following examples increment the reference count for the object in the variable `bee`.

```
bee='abcd'
# Initialize variable, reference count set to 1
ruf=bee
#Reference count incremented by assigning object to another variable
func(bee)
#Reference count incremented by calling a function
```

The reference count of an object is decremented when the execution of that function, method, or class instantiation is completed. All the objects used in a piece of code are destroyed after the execution of that piece of code is complete, which results in decrementing the reference count.

The reference count to an object is decremented when the variable containing that object is reassigned to another variable. For example,

```
ruf=beast
```

The reference count to an object is also decremented when the variable containing an object is deleted using the `del` statement. For example,

```
del ruf
```

The preceding `del` statement will decrement the reference count to the object `'abcd'` by `1` and delete the variable `ruf`. Furthermore, deleting the variable `bee` will delete the final reference to `'abcd'`; therefore, the object becomes inaccessible and becomes a part of garbage collection.

This chapter has explained the standard data types and their basic features. Let's now use them to help the sales department of Techsity University, which wants to calculate the total sales made on a particular day through its Web site.

Problem Statement

The sales department needs the daily sales report from its Web site for Thursday. On that day, five students purchased online courses from the Techsity Web site. You need

Table 2.5 Variables and Data Types Identified for the Problem Statement

NAMES OF CUSTOMERS	TOTAL PURCHASES (IN $)
Ken	234
William	200
Catherine	120.34
Steve	124.3
Mark	175

to display the names of these students. You also need to display the total purchases made by each student. The names of the students and the total purchases made are given in Table 2.5.

Task List

- **Create a sequence to store all the names of the students.**
- **Write the code to display the names of the students.**
- **Declare a dictionary of purchases made by students with the names of the students as the key.**
- **Write the code to display the purchases made by the students.**
- **Save and execute the code.**
- **Verify the details.**

Create a Sequence to Store All the Names of the Students

In the preceding problem, you do not have to add or delete any items from the sequence, so you can use either a list or a tuple to store the names of students.

```
stud_name=['Ken','William','Catherine','Steve','Mark']
```

Write the Code to Display the Names of the Students

The following `print` statements can be used to display the name of each student contained in `stud_name`.

```
print stud_name[0]
print stud_name[1]
print stud_name[2]
print stud_name[3]
print stud_name[4]
```

Declare a Dictionary of Student Purchases with the Names of the Students as the Key

You can declare the following dictionary, `stud_pur`, containing the name of each student as the key and his or her purchases as the value corresponding to each key.

```
stud_pur={'Ken':234.0,'William':200.0,'Catherine':120.34,'Steve':124.30,
'Mark':175.0}
```

Write the Code to Display the Student Purchases

Print the purchases made by each student in the following manner.

```
print 'The purchases made by',stud_name[0],'are',stud_pur[stud_name[0]]
print 'The purchases made by',stud_name[1],'are',stud_pur[stud_name[1]]
print 'The purchases made by',stud_name[2],'are',stud_pur[stud_name[2]]
print 'The purchases made by',stud_name[3],'are',stud_pur[stud_name[3]]
print 'The purchases made by',stud_name[4],'are',stud_pur[stud_name[4]]
```

Let's combine the preceding code snippets to create a complete code as follows:

```
stud_name=['Ken','William','Catherine','Steve','Mark']
print "Names of students who made purchases on Thursday are:"
print stud_name[0]
print stud_name[1]
print stud_name[2]
print stud_name[3]
print stud_name[4]
print '-'*40
stud_pur={'Ken':234.0,'William':200.0,'Catherine':120.34,
'Steve':124.30,'Mark':175.0}
print 'The purchases made by',stud_name[0],'are',stud_pur[stud_name[0]]
print 'The purchases made by',stud_name[1],'are',stud_pur[stud_name[1]]
print 'The purchases made by',stud_name[2],'are',stud_pur[stud_name[2]]
print 'The purchases made by',stud_name[3],'are',stud_pur[stud_name[3]]
print 'The purchases made by',stud_name[4],'are',stud_pur[stud_name[4]]
```

Save and Execute the Code

To be able to view the output of the above code, the following steps have to be performed:

1. Write the previous code in a text editor.
2. Save the file as **probstat2.py**.
3. Change the current directory to where you have saved the above file.
4. In the shell prompt, type:

```
$ python probstat2.py
```

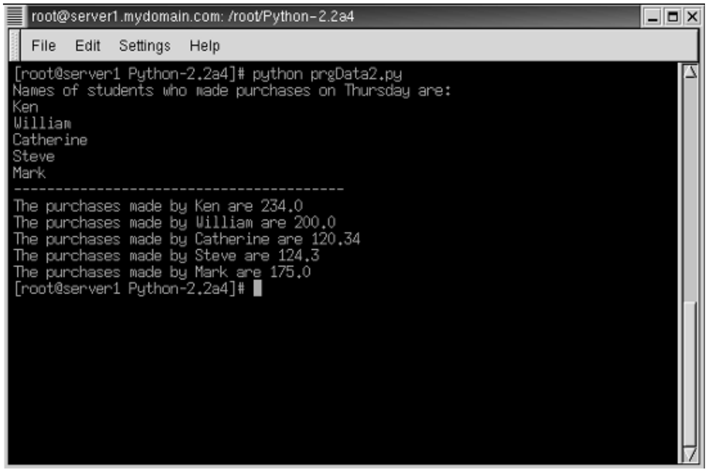


Figure 2.3 Output of the second problem.

Verify the Details

Verify whether all the values are displayed correctly and match Figure 2.3.

Summary

In this chapter, you learned the following:

- The `print` statement is used in Python to display data in Python.
- The `raw_input()` function is used to accept input from the user.
- A comment begins from the hash/pound sign and continues until the end of the line.
- The Python interpreter can act as a simple calculator. When you write an expression in it, it returns a value. The expressions used in the Python interpreter are the same as the ones in most programming languages, such as C, C++, and Pascal. The operators used are also the same, such as `+`, `-`, `*`, and `/`.
- Python uses objects for data abstraction. Any entity that contains any type of value or data is called an object. You can classify all data as an object or as a relation to an object. Each object has three characteristics: identity, type, and value.

  Identity. The identity of an object refers to the address of the object in the memory. It is also a unique identifier that makes it distinct from the rest of the objects. You can use the `id()` built-in function to obtain the memory address of a variable.

Type. The type of an object determines the operations that are supported by an object. It also defines the values that are possible for the objects of that type and the operations that can be performed on that object. The `type()` built-in function can be used to determine the type of the Python object.

Value. The value of an object refers to the data item contained in that object.

- Objects whose values can be changed after they are created are called *mutable* objects. Objects whose values cannot be changed after they are created are called *immutable* objects.
- Variables are reserved memory locations to store values.
- Python variables do not have to be explicitly declared. The declaration happens automatically when you assign a value to a variable. You can assign a value to a variable by using the = sign.
- Python declares the type and memory space required by a variable at run time. The interpreter also manages memory itself. When a variable is not used any longer—that is, when it is not using the memory—it is reclaimed to the system. This mechanism is called *garbage collection*. Garbage collection is done by tracking the number of references made to an object. This is referred to as *reference counting*.
- Python has five standard data types:
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary
- Python uses the following types of numbers:
  - Regular or plain integer
  - Long integer
  - Floating-point real number
  - Complex number
- The arithmetic operators available in Python are, +, -, *, /, //, **, and %.
- A string can be defined by enclosing characters within single or double quotes.
- The method of extracting a single character or a substring from a string by using an index or indices is called *slicing*. When using slice notation, two indices that are separated by a colon can be used to extract substrings.
- A list contains items separated by commas enclosed within square brackets (`[]`). All the items belonging to a list need not be of the same data type. Like strings, indices can be used to extract values from lists; however, lists are mutable data types.

- A tuple consists of a number of values separated by commas; however, unlike lists, tuples are enclosed within parentheses. Tuples are immutable data types. You can use indices or slices, though, to access the values in a tuple.
- In a dictionary, values are mapped according to keys. The key does not need to be a numeric value to index a data item; it can be any immutable data type, such as strings, numbers, or tuples. A tuple can be used as a key only if it does not contain any mutable object directly or indirectly. In other words, the Python dictionary is an unordered set of *key:value* pairs.
- Every programming language defines a set of rules to build variable names. Such names are called *identifiers*. Among all names allowed in Python, certain identifiers are reserved by the language and cannot be used as programmer-defined identifiers. These names are called *keywords*.

CHAPTER

3

Intrinsic Operations and Input/Output

OBJECTIVES:

In this chapter, you will learn to do the following:

- Use more methods of accepting user input
- Format the output:
  - The % operator
  - The special characters
  - The raw string operator
- Use intrinsic operations:
  - For numeric data types
  - For strings
  - For lists and tuples
  - For dictionaries

Getting Started

In the previous chapter, we familiarized you with the syntax and the standard data types used in Python. You also learned how to perform basic operations by using these data types. You identified two ways of displaying the output, using the `print` statement and using variables and expressions directly at the interpreter. That, however, is not how you may always want to display the output. You might want a definite spacing between each element, or you may need to print special characters that you cannot do using these ways. In this chapter, you will learn how to format the output to enhance its visual appeal. This chapter will further discuss the built-in functions that you can use with each data type. You might require an input from a user in different ways, or you might print an output by formatting it. Python provides many features that enable you to accomplish these tasks. Let's consider a scenario that will require the use of these features to enhance your learning.

Using Input/Output Features and Intrinsic Operations for Data Types in Python

Problem Statement

The data entry operator of Techsity University is asked to enter the details of the courses. He wants the data he enters to be displayed on the screen for verification. Prepare an output screen for him. The details he has to enter for each course are as follows:

- Course code
- Course title
- Course duration
- Course fee
- Start date
- End date
- Number of seats

He also wants additional information about the year in which a student will pass out, based on the end date of the course.

Let's identify the relevant tasks that will help you solve the problem.

Task List

- **Identify the variables and data types to be used.**
- **Write the code to accept and display the details.**
- **Execute the code.**

Table 3.1 Variables and Data Types Identified for the Problem Statement

VARIABLE NAME	DATA TYPE
course_code	String
course_title	String
course_dur	Integer
course_fee	Float
start_date	String
end_date	String
no_of_seats	Integer

Identify the Variables to Be Used

Based on knowledge from the previous chapter, the variables shown in Table 3.1 can be identified.

Let's now identify the tools that will further equip you to find a solution to the problem statement.

Accepting User Input

You already learned about the built-in `raw_input()` function that prompts the user with a string, accepts an input, and stores the input in the form of a string. For example,

```
>>> string=raw_input('Enter a string ')
Enter a string Daniel
>>> string
'Daniel'
```

The preceding statements accept a string from the user and store the input in the variable `string`. Consider the following example:

```
>>> ls=raw_input('Enter a list ')
Enter a list ['aaa',1234]
>>> ls
" ['aaa',1234]"
>>> type(ls)
<type 'str'>
```

In this example, notice that even if you enter a list object as the input, the value stored by the `raw_input()` function in the variable `ls` is in the form of a string object. To solve this problem, Python provides the `input()` built-in function. The

raw_input() function accepts the user input in the supplied form and stores the input in the form of a string. The input() function first evaluates the user input, determines if the user input is an expression, evaluates the type, and then stores it in a variable. While storing the object of the data, the input function does not change the type of the object to a string. For example,

```
>>> inp=input('Enter a list')
Enter a list ['aaa',1234]
>>> inp
['aaa', 1234]
>>> type(inp)
<type 'list'>
```

Formatting the Output

In the previous chapter, you learned about some special characters that can be used in the print statement to alter the way a string is displayed. These were backslash (\) to include quotes in a string, backslash and n (\n) to escape new lines, and triple quotes (''',""") to print a string in the original form. Let's learn about some more formatting features in Python, such as the % operator, special characters, and the raw string operator.

The % Operator

The way the % operator works in Python is similar to the printf() function in C. It also supports the printf() formatting codes. The syntax for using the % operator is this:

```
print_string % (convert_arguments)
```

In the syntax, print_string is the string that has to be printed as is. It contains % codes, each of which matches the corresponding argument in convert_arguments. Although the % operator converts the arguments according to the % codes supplied in print_string, the outcome is always a string. In other words, the result of the formatting is printed in the form of a string. Table 3.2 lists various % codes.

Table 3.2 % Codes and Their Conversion

% CODE	CONVERSION
%c	Converts to a character
%s	Converts to a string and applies the str() function to the string before formatting
%i	Converts to a signed decimal integer

% CODE	CONVERSION
`%d`	Converts to a signed decimal integer
`%u`	Converts to an unsigned decimal integer
`%o`	Converts to an octal integer
`%x`	Converts to a hexadecimal integer with lowercase letters
`%X`	Converts to a hexadecimal integer with uppercase letters
`%e`	Converts to an exponential notation with lowercase "`e`"
`%E`	Converts to an exponential notation with uppercase "`e`"
`%f`	Converts to a floating-point real number
`%g`	Converts to the value shorter of `%f` and `%e`
`%G`	Converts to the value shorter of `%f` and `%E`

Hexadecimal Conversion

Let's consider a few examples of hexadecimal conversions by using the `%` operator.

```
>>> "%x" % 255
'ff'
>>> "%X" % 255
'FF'
```

The previous examples use the hexadecimal conversion code with the `%` operator. Note that when used with lowercase `x`, the `%` operator generates the output in lowercase letters and when used with uppercase `X`, the `%` operator generates the output in uppercase letters.

Floating-Point and Exponential Notation Conversion

Consider a few examples of floating-point and exponential conversions by using the `%` operator.

```
>>> '%f' % 676534.32143
'676534.321430'
>>> '%.2f' % 676534.32143
'676534.32'
```

Note that if `.2` is used before `f`, two digits are printed after the decimal point.

```
>>> '%12.2f' % 676534.32143
'   676534.32'
```

In the preceding example, we specified the minimum total width of the output as 12 and the digits after the decimal point as 2. Therefore, the output has 9 digits to display and is padded with 3 spaces in the beginning.

```
>>> '%e'% 1332.234445
'1.332234e+003'
>>> '%E' % 1332.234445
'1.332234E+03'
>>> '%f' % 1332.234445
'1332.234445'
>>> '%g' % 1332.234445
'1332.23'
```

Integer and String Conversion

Consider a few examples of integer and string conversions by using the `%` operator.

```
>>> '%d' % 65
'65'
>>> '%3d' % 65
' 65'
>>> '%-3d' % 65
'65 '
```

In the preceding example, we used a number with `%d`. This is to specify the minimum total width of the converted string from the integer. If the number specified is more than the number of digits in the number, the string is padded with spaces in the beginning. Specifying a negative number with `%d` pads the spaces to the right of the printed number. For example,

```
>>> '%-4d' % 65
'65  '
```

Specifying a zero along with the specified number pads the string with zeros instead of spaces. For example,

```
>>> '%03d' % 65
'065'
```

Using `+` with the `%` operator displays the `+` sign with the number passed in `convert_argument` if it is a positive number.

```
>>> '%+d' % 65
'+65'
>>> '%+d' % -65
'-65'
```

The conversion can also be performed in a statement. Here's an example of `%s` for a string conversion in a `print` statement.

```
>>> print "Your registration number is: %s" % 'A001'
Your registration number is: A001
```

You can also pass multiple arguments to the % operator as a tuple. For example,

```
>>> print "The name for the registration number %s is %s" \
... %('A001','John')
The name for the registration number A001 is John
>>> print "%s has opted for %d courses" % ('Steve',5)
Steve has opted for 5 courses
>>> print "mm/dd/yy:%02d/%02d/%02d" % (2,13,1)
mm/dd/yy:02/13/01
```

Following is an example in which you can use a dictionary as an argument for the % operator.

```
>>> 'The domain name in the host name %(Host)s is %(Domain)s' %\
... {'Domain':'ucla','Host':'Test12.mktg.ucla.edu'}
'The domain name in the host name Test12.mktg.ucla.edu is ucla'
```

Special Characters

You might want the strings to include characters, such as tabs and quotes. To do this, use some single characters paired with a backslash (`\`). These single characters along with a backslash denote the presence of a special character. Let's consider an example of the NEWLINE character (`\n`).

```
>>> print "Hello \nworld!!!"
Hello
world!!!
```

Note that `\n` is not printed in the output. There is also a new line before the beginning of the second word. In addition, when you have a long statement that exceeds a single line, you can use a backslash to escape NEWLINE for continuing the statement. For example,

```
>>> print "The name for the registration number %s is %s" \
... %("A002","Steve")
The name for the registration number A002 is Steve
```

To print special characters, you use not only the characters with a backslash but also their decimal, octal, and hexadecimal values. For example,

```
>>> print "Hello \012world!!!"
Hello
world!!!
```

Table 3.3 lists various backslash escape characters along with their octal, decimal, and hexadecimal equivalents.

Table 3.3 Escape Characters for Strings

ESCAPE CHARACTER	NAME	CHARACTER	DECIMAL	OCTAL	HEXADECIMAL
`\n`	Newline\ Linefeed	LF	10	012	0x0A
`\t`	Horizontal Tab	HT	9	011	0x09
`\b`	Backspace	BS	8	010	0x08
`\0`	Null character	NUL	0	000	0x00
`\a`	Bell	BEL	7	007	0x07
`\v`	Vertical tab	VT	11	013	0x0B
`\r`	Carriage return	CR	13	015	0x0D
`\e`	Escape	ESC	27	033	0x1B
`\"`	Double quote	"	34	042	0x22
`\'`	Single quote	'	39	047	0x27
`\f`	Form feed	FF	12	014	0x0C
`\\`	Backslash	\	92	134	0x5C

The Raw String Operator

Consider a situation in which you actually want to print `\t` in the output by using the `print` statement. The moment the Python interpreter encounters an escape character, the interpreter converts it into a special character. To counter this behavior, Python provides the raw string operator, uppercase or lowercase `r`. When preceded by an `r`, a string is converted to a raw string. For example,

```
>>> s=r'Hello\n'
>>> print s
Hello\n
```

REGULAR EXPRESSIONS

The raw strings feature that enables the interpreter to counter the behavior of special characters is useful when composing Regular Expressions. Regular Expressions are special strings that are used to define special search models for strings. They contain special symbols to denote characters, variable names, character classes, and criteria according to which characters are grouped and matched. Regular Expressions also contain the symbols that denote escape sequences. Therefore, using raw strings helps avoid the confusion between the escape sequences and characters that are a part of Regular Expressions.

Introduction to Intrinsic Operations

Intrinsic operations are built into the standard libraries in Python. These operations can be performed on Python objects including standard data types. We already discussed some operations involving the standard data types. These include variable assignments, forming expressions by using variables and operators, and some standard built-in functions. We briefly introduced the `type()` and `id()` functions in the previous chapter. Let's see how the `id()` and `type()` functions can be used to extract the type and identity of an object. For example,

```
>>> type('abcd')
<type 'str'>
>>> type(0xdd)
<type 'int'>
>>> a=9+4j
>>> type(a)
<type 'complex'>
```

In the preceding examples, the `type()` function returns the type of the object passed to it as the argument. The `id()` function can be used to return the memory address of an object. For example,

```
>>> a=9+4j
>>> id(a)
9639792
>>> a=a+76
>>> id(a)
7965324
```

Notice that because number is an immutable data type, changing the value of a variable that contains the number creates another object with another memory address assigned to that object.

```
>>> a=[31,'ddd']
>>> id(a)
17429076
>>> a.append('abcd')
>>> id(a)
17429076
```

Notice that in the previous examples, because list is a mutable data type, changing the items in a list does not change the memory address of the list.

Another function that can be used on all data types is the `cmp()` function. The syntax of the `cmp()` built-in function is this:

```
cmp(ob1,ob2)
```

The `cmp()` function compares two Python objects, `ob1` and `ob2`, and returns `0` if `ob1` equals `ob2`, `1` if `ob1` is greater than `ob2`, and `-1` if `ob1` is less than `ob2`.

```
>>> a,b=-8,13
>>> cmp(a,b)
-1
>>> hex(34)
'0x22'
>>> cmp(0x22,34)
0
```

In the first example, an integer is compared with another integer, and therefore it returns `-1` because `ob1` is less than `ob2`. In the second example, the hexadecimal value of 34 is compared with 34. The result is `0` because both the numbers are converted to the same form before comparison and therefore evaluate to the same value. In other words, numeric data types are compared according to their numeric value.

Sequence objects, such as strings, lists, tuples, and dictionaries, can be compared with other objects. The comparison is made using *lexicographical ordering*. This means that first the Python interpreter compares the first two items. If they are different, the output is determined based on this comparison. If they are identical, the next two items are compared. If they differ, the output is determined by this comparison. This continues until the last item in the sequence is exhausted.

```
>>> cmp((1,2,3,4),(1,2,4))
-1
>>> a,b='abc','pqr'
>>> cmp(a,b)
-1
```

If all the items in the two sequences are equal, the sequences are considered equal. If the first few items of one sequence are the same as in the other sequence, the smaller sequence is considered to be less than the longer one.

```
>>> cmp([123,'abc',888],[123,'abc'])
1
```

If the sequences being compared contain other sequences as their data items, the comparison is made recursively.

```
>>> cmp(['abcd',(123,'abc',555)],['abcd',(123,'xyz',897)])
-1
```

You can also compare objects of different types. The types are compared by their names. Therefore, dictionaries are smaller than lists, lists are smaller than strings, and strings are smaller than tuples.

In addition to the functions that can be performed on all data types, there are functions that can be performed only on sequence types, such as the `len()`, `max()`, and `min()` functions. We learned that the `len()` function is used to find the length of a sequence. The `max()` and `min()` functions can be used to find the element with the minimum and maximum values, respectively. For example, when using a string, these functions return the highest and lowest characters, respectively.

```
>>> max('abc')
'c'
>>> min('abc')
'a'
```

Consider another example for lists as follows:

```
>>> list=[23,23.1,23L,234e-1]
>>> max(list)
23.399999999999999
>>> min(list)
23
```

When the elements of a sequence are of different types, each element is treated as a separate object that has to be compared lexicographically. The order of precedence for the standard data types is as follows:

dictionaries < lists< strings<tuples

```
>>> list_str=['jjj',445,['vf',23]]
>>> max(list_str)
'jjj'
>>> min(list_str)
445
```

After the brief introduction to the operations on data types, let's consider each data type individually.

Intrinsic Operations for Numeric Data Types

The operations on numeric data types can be classified into conversion functions and other operational functions. Table 3.4 lists the conversion functions that can be applied to numeric data types.

Table 3.4 Conversion Built-in Functions for Numeric Types

FUNCTION	DESCRIPTION	EXAMPLE
`int(ob)`	Converts a string or number object to an integer.	`>>>int('15')` `15`
`long(ob)`	Converts a string or number object to long.	`>>>long('12')` `12L`
`float(ob)`	Converts a string or number object to a floating-point number.	`>>> float(10)` `10.0`

`complex(string)` or `complex(real,imag)`	Converts a string to a complex number or takes a real number and an imaginary number (optional) and returns a complex number with those components.	`>>> complex('76')`
		`(76+0j)`
		`>>> complex(45,8)`
		`(45+8j)`

Python also provides a few operational functions for numeric data types. Table 3.5 lists the operational functions applicable for numeric types.

Table 3.5 Operational Functions for Numeric Types

FUNCTION	DESCRIPTION	EXAMPLE

`abs(ob)`	Converts the string or number object to its absolute.	`>>> abs(-13)`
		`13`
		`>>> abs(5.)`
		`5.0`

`coerce(ob1,ob2)`	Converts `ob1` and `ob2` to the same numeric type and returns the two numbers as a tuple.	`>>> coerce(12.0,8)`
		`(12.0, 8.0)`
		`>>> coerce(2,86L)`
		`(2L, 86L)`

FUNCTION	DESCRIPTION	EXAMPLE

`divmod(ob1,ob2)`	Divides `ob1` and `ob2` and returns both the quotient and remainder as a tuple. For complex numbers, the quotient is rounded off. Complex numbers use only the real component of the quotient.	`>>> divmod(10,6)`
		`(1, 4)`
		`>>> divmod(10.6,3.4)`
		`(3.0, 0.39999999999999991)`
		`>>> divmod(78,23l)`
		`(3L, 9L)`

`pow(ob1,ob2,mod)`	Raises `ob2` to the power of `ob1`. Takes an optional argument `mod`, divides the result by `mod`, and returns the remainder.	`>>> pow(2,3)`
		`8`
		`>>> pow(2,3,5)`
		`3`

`round(flt,dig)`	Rounds off the float `flt` to the `dig` digits after the decimal point and assumes `0` if `dig` is not supplied.	`>>> round(67.5324)`
		`68.0`
		`>>> round(4.46,1)`
		`4.5`

In addition to the built-in functions that are applicable for all numeric types, Python has some functions applicable only to integers. These functions can be classified into base and ASCII conversion functions.

You already know that Python supports the hexadecimal and octal representation of numbers. You can use the base conversion functions to convert an integer into its hexadecimal or octal equivalent. These functions are `hex()` and `oct()`. Both functions take an integer and return a corresponding hexadecimal or octal equivalent as a string.

```
>>> hex(35)
'0x23'
>>> hex(677)
'0x2a5'
>>> hex(4*789)
'0xc54'
>>> hex(45L)
'0x2DL'
>>> oct(863)
'01537'
>>> oct(6253915L)
'027666533L'
```

Python also provides functions to convert integers into their ASCII (American Standard for Information Interchange) characters and vice versa. Each character is mapped to a unique numeric value from 0 to 255, listed in a table called the ASCII table. The mapping remains the same for all machines using the ASCII table. The `ord()` function takes a single character and returns the ordinal value associated with that ASCII character. For example,

```
>>> ord('d')
100
>>> ord('D')
68
>>> ord('l')
108
```

To convert a value to its corresponding ASCII character, you can use the `chr()` function. For example,

```
>>> chr(65)
'A'
>>> chr(100)
'd'
>>> chr(108)
'l'
```

You saw how intrinsic operations on integers make it simple to handle programming tasks for which you might have to write long code. Let's learn some intrinsic operations possible for strings.

Intrinsic Operations for Strings

We discussed the `cmp()`, `max()`, and `min()` standard type functions, which perform lexicographic comparison for all types. We also discussed the `len()` sequence type function, which returns the length of a sequence. In addition, the `max()` and `min()` functions can be used to find the character with the minimum and maximum values, respectively. For example,

```
>>> max('abc')
'c'
>>> min('abc')
'a'
```

Often, you might need to convert a value of a particular data type into a string. Python allows you to do this in a number of ways.

The `repr()` function. You can pass an object of any data type to the `repr()` function to convert it to a string.

```
>>> astr=repr(76)
>>> astr
'76'
```

Recall that the presence of double or single quotes indicates that `astr` is a string.

```
>>> ls=[43,12.23]
>>> bstr=repr(ls)
>>> bstr
'[43, 12.23]'
>>>cstr='xyz'
>>> ls=[astr,cstr]
>>> ls_str=repr(ls)
>>> ls_str
"['76', 'xyz']"
```

The `str()` function. You can also pass the value to the `str()` function. For example,

```
>>> a='Flower \tred'
>>> b=78
>>> tup=(a,b)
>>> str(tup)
"('Flower \\tred', 78)"
```

Note that using the `str()` function to convert to a string adds backslashes if a backslash is already present. This happens regardless of the method you use to convert to a string.

```
>>> print str(tup)
('Flower \tred', 78)
```

As expected, the escape character and the backslash appear in the string when displayed using with the print statement and is not replaced with the corresponding special character, which is a horizontal tab in this case.

Reverse quotes (` `). You can write the value or variable in reverse quotes to convert it to a string. This method works for all data types except numbers.

```
>>> tup=('rep','tree')
>>> `tup`
"('rep', 'tree')"
>>> jo='welcome'
>>> string=`jo`
>>> string
"'welcome'"
```

Note that when you enclose a string within reverse quotes, string quotes are added to it.

```
>>> `5*30`
'150'
```

If the value enclosed in the reverse quotes is an expression, it is evaluated first and then converted into a string.

In addition to the functions discussed previously, Python also provides some common operations for strings in the form of methods. For example, the `capitalize()` method capitalizes the first character of a string. These methods can be called using a variable containing a string value.

```
>>> s='hello'
>>> s.capitalize()
'Hello'
```

Table 3.6 lists some of these methods for strings.

Table 3.6 String Type Built-in Methods

METHOD	EXPLANATION
`s.capitalize()`	Capitalizes the first letter of the string `s`.
`s.center(width)`	Centers the string in the length specified by width and pads the columns to the left and the right with spaces.
`s.count((sub[, start[, end]])`	Counts the number of occurrences of `sub` in the string `s` beginning from the `start` index and continuing until the `end` index. Both start and end indices are optional and default to `0` and `len(s)`, respectively, if not supplied.
`s.endswith(sub[, start[, end]])`	Returns `1` if the string s ends with the specified substring `sub`; otherwise returns `-1`. Search begins from the index `start` until the end of the string. Default is to start from the beginning and finish at the end of the string.
`s.expandtabs([tabsize])`	Returns the string after replacing all tab characters with spaces. If `tabsize` is not given, the tab size defaults to 8 characters.
`s.find(sub[, start[, end]])`	Returns the beginning index in the string where the substring `sub` begins from the `start` index and continues until the `end` index. Both start and end indices are optional and default to `0` and `len(s)` if not supplied.
`s.index(sub[, start[, end]])`	Similar to `find()` but raises an exception if the string is not found.

METHOD	EXPLANATION
`s.isalnum()`	Returns `1` if all the characters in the string `s` are alphanumeric and there is a minimum of one character; otherwise returns `0`.
`s.isalpha()`	Returns `1` if all the characters in the string `s` are alphabetic and there is a minimum of one character, otherwise returns `0`.
`s.isdigit()`	Returns `1` if all the characters in the string `s` are digits.
`s.islower()`	Returns `1` if all the alphabetic characters in the string are in lowercase and there is at least one alphabetic character; otherwise returns `0`.
`s.isspace()`	Returns `1` if there are only whitespace characters in the string and otherwise returns `0`.
`s.istitle()`	Returns `1` if the string is in title case. True only when uppercase characters follow lowercase characters and lowercase characters follow only uppercase characters. Returns false otherwise.
`s.isupper()`	Returns true if all the alphabetic characters in the string are in uppercase and returns false otherwise.
`s.join(seq)`	Returns a string that is the concatenation of the strings in the sequence `seq`. The separator between elements is the string `s`. The sequence `seq` should contain only strings.
`s.ljust(width)`	Returns a copy of string `s` left justified in the total number of columns equal to `width`. Extra width is padded by spaces. If the length of the string is greater than the width, it is not truncated.

Continues

Table 3.6 String Type Built-in Methods (Continued)

METHOD	EXPLANATION
`s.ljust(width)`	Returns a copy of string `s` right justified in the total number of columns equal to `width` without truncating the string. Extra width is padded by spaces.
`s.center(width)`	Returns a copy of string `s` centered in the total number of columns equal to `width` without truncating the string. Extra width is padded by spaces.
`s.lower()`	Returns a copy of the string converted to lowercase.
`s.upper()`	Returns a copy of the string converted to uppercase.
`s.swapcase ()`	Returns a copy of the string after converting uppercase characters to lowercase and vice versa.
`s.title ()`	Returns a copy of the string after converting the first letters of all the words to uppercase and the rest to lowercase.
`s.lstrip()`	Returns a copy of the string after removing the leading whitespaces.
`s.rstrip()`	Returns a copy of the string after removing the trailing whitespaces.
`s.strip()`	Returns a copy of the string after removing both leading and trailing whitespaces.
`s.replace(oldsub, newsub[, num])`	Replaces all occurrences of the substring `oldsub` in the string `s` with `newsub`. If the optional argument `num` is supplied, only the first `num` occurrences are replaced.
`s.rfind(sub [,start [,end]])`	Similar to `find()` except `rfind()`, searches the string backward.
`s.rindex(sub[, start[, end]])`	Similar to `rfind()` but raises `ValueError` when the substring `sub` is not found.

METHOD	EXPLANATION
`s.split([sep [,num]])`	Returns a list of substrings in the string `s`, which is separated by `sep` as the delimiter string. If `num` is supplied, maximum `num` splits are performed. If `sep` is either not specified or None, the whitespaces are treated as separators.
`s.splitlines([keepends])`	Returns a list of the lines in the string, breaking at the end of lines. Line breaks are not included in the resulting list if `keepends` is specified to be `0` (false).
`s.startswith(prefix[,start[,end]])`	Returns `1` if string starts with the `prefix`, otherwise returns `0`. If start and end are specified, the search begins from the `start` index and `finishes` at the end index. If not specified, the search starts from the beginning and ends at the last character of the string.
`s.translate (table[, deletechars])`	Returns a copy of the string where all characters in the optional argument `deletechars` are removed and the remaining characters are mapped through the given translation table, which must be a string (256 characters).

Let's look at some examples using the methods mentioned in Table 3.6.

```
>>> strpy='python is my choice'
>>> strpy.title()
'Python Is My Choice'
>>> strpy.center(30)
'     python is my choice      '
>>> strpy.find('oi',6)
15
>>> strpy.isalpha()
0
>>> strpy.replace(' ',':',2)
'python:is:my choice'
>>> strpy.startswith('th',2,9)
```

```
1
>>> strpy.split()
['python', 'is', 'my', 'choice']
>>> '='.join(['name','steve'])
'name=steve'
```

The `join()` and `split()` methods can be used when you want to first split each word in a line and then join it again using a separator. For example,

```
>>> ':'.join(strpy.split())
'python:is:my:choice'
```

You can also perform the same task by using the `replace()` method as follows:

```
>>>strpy.replace(' ',':')
'python:is:my:choice'
```

Intrinsic Operations for Lists and Tuples

The basic operations that can be performed with lists, such as assigning values to lists, inserting items, removing items, and replacing items, were discussed in the previous chapter. In this chapter, let's learn more about lists. You are aware that lists and tuples offer similar features of slicing and indexing except that lists are mutable and tuples are not. You might wonder why Python needs two similar kinds of data types. Consider an example to answer this question. There may be a situation in which you are calling a function by passing data to it. If the data is sensitive, you want it to remain secure and not be altered in the function. In such a situation, tuples are used, which are mutable and cannot be altered in the function. Lists, though, are best suited for a situation in which you are handling dynamic data sets, which allow elements to be added and removed as and when required. Python also allows you to convert lists to tuples and vice versa, rather painlessly, by using the `tuple()` and `list()` functions, respectively. These functions do not actually convert a list into a tuple or vice versa. They create an object of the destination type containing the same elements as that in the original sequence. For example,

```
>>> listvar=['abcd',123,2.23,'efgh']
>>> tupvar=tuple(listvar)
>>> tupvar
('abcd', 123, 2.23, 'efgh')
>>> id (tupvar)
15802876
>>> id (listvar)
17426060
```

Notice that the identity of `listvar` is different from the identity of the converted list `tupvar`. Note also that `listvar` is a list with the items enclosed in `[]` and that `tupvar` is a tuple with its arguments enclosed in `()`. Similarly, a tuple can also be converted to a list by using the `list()` function. For example,

Table 3.7 List Type Built-in Methods

METHOD	EXPLANATION
`s.append(ob)`	Adds the object `ob` at the end of the list.
`s.extend(seq)`	Appends the items in the sequence `seq` to the list.
`s.count(ob)`	Counts the number of occurrences of the object `ob` in the list.
`s.index(ob)`	Returns the smallest index in the list where the object `ob` is found.
`s.insert(i,ob)`	Inserts the object `ob` at the ith position in the list.
`s.pop([i])`	Returns the object at the ith position or the last position from the list, if not specified. It also removes the item returned from the list.
`s.remove(x)`	Removes the object `ob` from the list.
`s.reverse()`	Reverses the items of the list.
`s.sort([func])`	Sorts the items in the list and uses the compare function `func`, if specified.

```
>>> tup1=(123,'abc',345)
>>> id(tup1)
17351292
>>> list1=list(tup1)
>>> id(list1)
17454260
```

Like strings, Python also provides some methods for lists to perform common operations on lists, such as adding, sorting, deleting, and reversing items. Table 3.7 lists some of these methods for lists. Because tuples are immutable data types, these methods do not apply to tuples.

Let's present some examples by using the methods mentioned in Table 3.7.

```
>>> listvar=['abcd',123,2.23,'efgh']
>>> listvar.append(45)
>>> listvar
['abcd', 123, 2.23, 'efgh', 45]
>>> listvar.remove(2.23)
>>> listvar
['abcd', 123, 'efgh', 45]
>>> listvar.insert(4,'elite')
>>> listvar
['abcd', 123, 'efgh', 45, 'elite']
>>> listvar.insert(1,'elite')
```

```
>>> listvar
['abcd', 'elite', 123, 'efgh', 45, 'elite']
>>> listvar.remove('elite')
>>> listvar
['abcd', 123, 'efgh', 45, 'elite']
>>> listvar.reverse()
>>> listvar
['elite', 45, 'efgh', 123, 'abcd']
>>> listvar.sort()
>>> listvar
[45, 123, 'abcd', 'efgh', 'elite']
```

In the preceding examples, we appended an item at the end of the list `listvar` and removed an object by specifying a value in the `remove()` method. Next, we inserted `'elite'` at the fourth position and then at the first position in the list. Note that the `remove()` method removes only the first occurrence of the object `'elite'` from the list. The `reverse()` and `sort()` functions reverse and sort the items in the list, respectively.

You learned about the intrinsic operations that can be performed on lists. Due to the mutability feature of lists, they are very flexible; other data structures can be built on lists very easily. Lists can also function like stacks and queues. Let's see how.

Lists as Stacks

You know of a stack as a pile of items, such as books or cards, in which the last item you place is the first one that can be removed. This is exactly what a stack means in terms of programming. The method of adding items to a stack is called "last-in, first-out" (LIFO). A list can also be used easily as a stack. The last item added to a list can be the first element to be retrieved. An item can be added to a list by using the `append()` method. The last item can be removed from the list by using the `pop()` method without passing any index to it. Consider the following example:

```
>>> stack=['a','b','c','d']
>>> stack.append('e')
>>> stack.append('f')
>>> stack
['a', 'b', 'c', 'd', 'e', 'f']
>>> stack.pop()
'f'
>>> stack
['a', 'b', 'c', 'd', 'e']
>>> stack.pop()
'e'
>>> stack
['a', 'b', 'c', 'd']
```

Notice that when you use lists in this manner, the last element added is extracted first and the element added before the last is extracted next.

Lists as Queues

A list can also be used easily as a queue. In a queue, the first item added to a list can be the first element to be retrieved. The method of adding items to a queue is called "first-in, first-out" (FIFO). An item can be added to a list by using the `append()` method. The last item can be extracted from the list by using an index of `0` in the `pop()` method. Consider the following example:

```
>>> queue=['a','b','c','d']
>>> queue.append('e')
>>> queue.append('f')
>>> queue
['a', 'b', 'c', 'd', 'e', 'f']
>>> queue.pop(0)
'a'
>>> queue.pop(0)
'b'
>>> queue
['c', 'd', 'e', 'f']
```

The `range()` function

Lists allow the use of another function, `range()`, which creates a list containing an arithmetic progression. This is useful when you need to iterate over a sequence of numbers. Iteration is performed using a looping statement. Chapter 4, "Programming Basics," discusses looping statements in detail. Here are a few examples of values returned by the `range()` function.

```
>>>range(7)
[0, 1, 2, 3, 4, 5, 6]
```

Notice that the list returned by the `range()` function does not contain the value passed to it as the last item. `range(7)` returns exactly seven values in the list, starting from the first legal index of the sequence of length 7. You can also specify the sequence generated by the `range()` function to start from a different index or specify a different increment. For example,

```
>>> range(3,7)
[3, 4, 5, 6]
```

In the preceding example, the resulting sequence starts from the first argument passed to the `range()` function and ends at one less than the second argument passed.

```
>>> range(-10,-100,-40)
[-10, -50, -90]
```

In the preceding example, the third argument passed is the number by which the values in the resulting list are incremented. Note that the items in the list differ by `-40`.

Intrinsic Operations for Dictionaries

You learned how a dictionary consists of key:value pairs and how each value can be addressed by using a key in a dictionary. Like strings and lists, Python also provides some built-in methods for dictionaries. Table 3.8 lists the methods available for dictionaries.

Here are a few examples of using dictionary methods.

```
>>> dict1={'name':'mac','ecode':6734,'dept':'sales'}
>>> dict1.values()
[6734, 'sales', 'mac']
>>> dict1.items()
[('ecode', 6734), ('dept', 'sales'), ('name', 'mac')]
>>> dict1.get('ecode')
6734
>>> dict1.has_key('dept')
1
>>> dict1.has_key('salary')
0
```

Table 3.8 Dictionary Type Built-in Methods

METHOD	EXPLANATION
`dict.clear()`	Deletes all the elements in the dictionary `dict`.
`dict.items()`	Returns a list of key:value pairs in the dictionary `dict` in the form of tuples.
`dict.keys()`	Returns a list of keys in the dictionary `dict`.
`dict.values()`	Returns a list of values in the dictionary `dict`.
`dict.has_key(key)`	Returns `1` if `key` is in the dictionary `dict`; otherwise returns `0`.
`dict.get(key,default)`	Returns the value for `key` or the value `default` if `key` is not found in the dictionary.
`dict.setdefault(key,default)`	Similar to `get()` but sets the value associated with `key` to `default`; `None` if `default` is not specified.
`dict.copy()`	Creates a copy of the object in `dict`.
`dict.update(dict2)`	Adds the values in the dictionary `dict2` to `dict`.

In the preceding example, the dictionary `dict1` is created and the values in it are extracted. The `items()` method is used to return the tuples of the key:value pairs in the dictionary, and the `get()` method is used to extract the value associated with a key. In the end, the `has_key` method is used to find out if a key exists in the dictionary.

Write the Code

Based on the preceding discussion, the code for the problem statement in the beginning of the chapter is as follows:

```
#Accept values
course_code=raw_input('Enter course code:')
course_title=raw_input('Enter course title:')
course_dur=input('Enter course duration (in hrs.):')
course_fee=float(input('Enter course fee (in $):'))
start_date=raw_input('Enter course start date (mm/dd/yy):')
end_date=raw_input('Enter course end date (mm/dd/yy):')
no_of_seats=input('Enter no. of seats:')
#Display the output
print
print '%-20s %-20s %-20s %-20s'% ('Course Code:',course_code,\
'Course Title:',course_title.title())
print '%-20s %-20d %-20s %-17.2f'% ('Course Duration:',\
course_dur, 'Course Fee:',course_fee)
print '%-20s %-20s %-20s %-20s'% ('Start Date:',start_date,\
'End Date:',end_date)
print '%-20s %-20d '% ('No. of seats:',no_of_seats)
ls=end_date.split('/')
print'\n'*3
print 'The year of passing out will be', ls[2]
```

Execute the Code

To be able to view the output of the preceding code, the following steps have to be executed:

1. Type the code in a text editor.
2. Save the file as **prgIntroper.py**.
3. Make the directory in which you saved the file the current directory.
4. On the shell prompt, type:

   ```
   $ python prgIntroper.py
   ```

Use Figure 3.1 as a sample to enter the input.
Figure 3.2 shows the sample output.

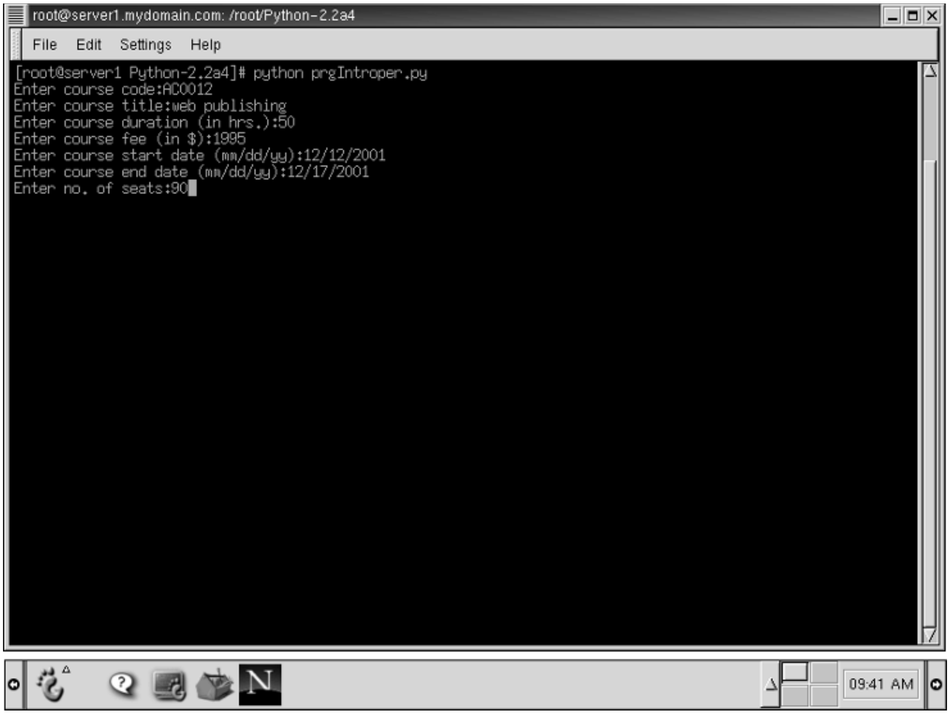


Figure 3.1 The sample input.

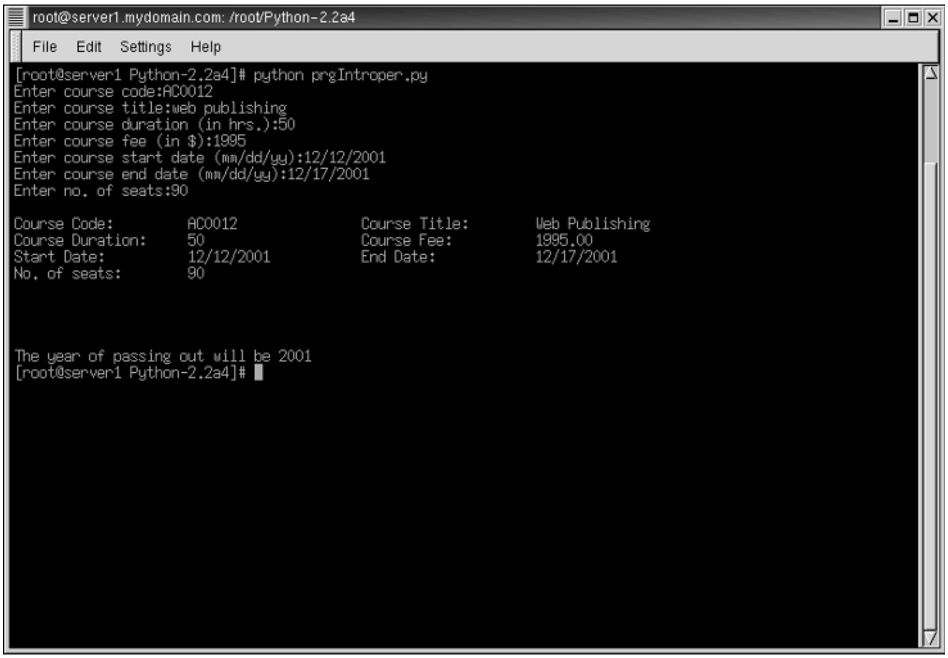


Figure 3.2 The sample output.

Summary

In this chapter, you learned the following:

- The `input()` function first evaluates the user input and its type and then stores it in a variable. While storing the object of the data, the input function does not change the type of the object to a string.
- The way the `%` operator works in Python is similar to the `printf ()` function in C. It also supports the `printf()` formatting codes. The syntax for using the `%` operator is this:

  ```
  print_string % (convert_arguments)
  ```

- Backslash escape characters can be used to print special characters that otherwise cannot be included in a string.
- When preceded by the raw string operator, uppercase or lowercase `r`, a string is converted to a raw string.
- Intrinsic operations are built into the Python standard libraries and can be performed on data types.
- The `id()` function can be used to return the memory address of an object.
- The `cmp()` function compares two Python objects, `ob1` and `ob2`, and returns `0` if `ob1` equals `ob2`, `1` if `ob1` is greater than `ob2`, and `-1` if `ob1` is less than `ob2`. The syntax of the `cmp()` built-in function is as follows:

  ```
  cmp(ob1,ob2)
  ```

- The operations on numeric data types can be classified into conversion functions and other operational functions.
- Any object containing a value can be converted to a string by using the following ways:
  - The `repr()` function
  - The `str()` function
  - Reverse quotes (`` ` ` ``)
- Python also provides some more common operations for strings in the form of methods.
- Python allows you to convert lists to tuples and vice versa by using the `tuple()` and `list()` functions, respectively.
- Like strings, Python also provides some methods for lists to perform common operations on lists, such as adding, sorting, deleting, and reversing items.
- A list can also be easily used as a stack. The last item added to a list is the first element to be retrieved. The method of adding items to a stack is called "last-in, first-out" (LIFO).

- A list can also be easily used as a queue. In a queue, the first item added to a list is the first element to be retrieved. The method of adding items to a queue is called "first-in, first-out" (FIFO).
- The `range()` function creates a list containing an arithmetic progression.
- Python also provides some built-in methods for dictionaries.

CHAPTER

4

Programming Basics

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Use the following conditional constructs:**
  - `if`
  - `if...else`
  - `elif`
  - `nested if`
- **Use the following loop constructs:**
  - `while`
  - `for`
- **Use the following statements with loops:**
  - `break`
  - `continue`

Getting Started

In the previous chapters, you learned about data types and variables and the intrinsic operations performed on them. While programming, however, you need to use objects, variables, and expressions in a clause that allows them to be executed after performing a check. There are situations in which you may want to reference data items repeatedly, perform operations on variables only when a certain condition holds true, or perform different operations for different values of the same variable. Programming constructs come in handy in such situations when you have to make choices or perform certain actions based on whether a particular condition holds true. In this chapter, you will use programming constructs, such as `if...else`, `elif`, `while`, `for`, `break`, and `continue`, and `pass` statements.

The conditions used in programming constructs usually resolve to either true or false. Conditions contain operands and conditional operators. Before we learn about programming constructs, let's understand the various types of conditional operators available in Python.

Conditional Operators

Conditional operators are used to compare values and test multiple conditions. The various types of conditional operators used to perform operations in Python are these:

- Comparison operators
- Boolean logical operators
- Bitwise operators
- Membership operators (used for sequence only)
- Identity operators

Comparison Operators

Comparison operators, when used in an expression, evaluate to an integer value, `1`, when the expression resolves to true and `0` when an expression resolves to false. Table 4.1 describes the various comparison operators.

The Python interpreter uses the following rules for comparison operators:

- Arithmetic rules are used for comparison between numbers.
- Strings, lists, and tuples are compared lexicographically by matching the ASCII value of each element in one sequence with that of the corresponding element in the other sequence.
- Comparisons for dictionaries are also done lexicographically by matching sorted lists of key:value pairs.

Table 4.1 Comparison Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`==`	Evaluates whether the operands are equal.	`x==y`	Returns `1` if the values are equal and `0` otherwise.
`!= or <>`	Evaluates whether the operands are not equal.	`x!=y`	Returns `1` if the values are not equal and `0` otherwise.
`>`	Evaluates whether the left operand is greater than the right operand.	`x>y`	Returns `1` if x is greater than y and `0` otherwise.
`<`	Evaluates whether the left operand is less than the right operand.	`x<y`	Returns `1` if x is less than y and `0` otherwise.
`>=`	Evaluates whether the left operand is greater than or equal to the right operand.	`x>=y`	Returns `1` if x is greater than or equal to y and `0` otherwise.
`<=`	Evaluates whether the left operand is less than or equal to the right operand.	`x<=y`	Returns `1` if x is less than or equal to y and `0` otherwise.

A few examples of using comparison operators follow:

```
>>> 32<50
1
>>> a=45
>>> b=15*3
>>> a==b
1
>>> a<=b
1
>>> a==b>50
0
>>> (1,2,3)<(1,2,4)
1
>>> 'aaa'>'abc'
0
```

Boolean Operators

You can use Boolean operators to combine the results of Boolean expressions. Table 4.2 describes the various Boolean logical operators.

Table 4.2 Boolean Operators

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
`and`	Evaluates to false if the first expression evaluates to false; otherwise, if the first expression evaluates to true, the `and` operator evaluates to the value of the second expression.	`x>5 and y<10`	The result is true if condition1, `x>5`, and condition2, `y<10`, are both true. If one of them is false, the result is false.
`or`	Evaluates to true if the first expression evaluates to false; otherwise, if the first expression evaluates to true, the `or` operator evaluates to the value of the second expression.	`x>5 or y<10`	The result is true if either condition1, `x>5`, or condition2, `y<10`, or both, evaluate to true. If both the conditions are false, the result is false.
`not`	Evaluates to true if its argument is false and false otherwise.	`not x>5`	The result is true if condition is false and false if condition is true.

The values returned by the operators `and` and `or` are not restricted to `0` or `1`. Both these operators return the last evaluated value. For example,

```
>>> x,y=45,65
>>> a,b='abc','xyz'
>>> (x<y) and (a,b)
('abc', 'xyz')
```

In the preceding example, the first expression is evaluated first. It is clear that the first expression resolves to true. Therefore, the `and` operator evaluates to the value of the second expression, which is a tuple.

Bitwise Operators (Integer-Only)

Data is stored internally in binary format (in the form of bits). A bit can have a value of `1` or `0`. Bitwise operators are used to compare integers in their binary formats.

Table 4.3 summarizes the details of bitwise operators.

Table 4.3 Bitwise Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`~(NOT)`	Evaluates to a binary value after a bitwise `NOT` on the operands.	`~x`	Results in -(x+1).
`& (AND)`	Evaluates to a binary value after a bitwise `AND` on the operands.	`x&y`	Bitwise `AND` results in a `1` if both the bits are `1`; any other combination results in a `0`.
`|(OR)`	Evaluates to a binary value after a bitwise `OR` on the two operands.	`x|y`	`OR` results in a `0` when both the bits are `0`; any other combination results in a `1`.
`^(XOR)`	Evaluates to a binary value after a bitwise `XOR` on the two operands.	`x^y`	`XOR` results in a `0` if both the bits are of the same value and `1` if the bits have different values.
`<<(Left shift)`	Left expression is shifted to the left by number of bits in the right expression.	`x<<y`	Multiplies `x` by 2^y.
`>>(Right shift)`	Left expression is shifted to the right by number of bits in the right expression.	`x>>y`	Divides `x` by 2^y.

In the examples shown in Table 4.3, `x` and `y` are integers and can be replaced with expressions. When you are applying bitwise operators, keep the following facts in mind:

- Negative numbers are treated as their binary complement value.
- When performing a bitwise comparison of two numbers, each bit of one number is compared with the corresponding bit of the other. This evaluates to a binary number. The result is the binary number converted to its decimal value. For example, the bitwise `and` operation between binaries 1011 and 1001 will result in 1001, as shown in the following representation:

1	0	1	1
1	0	0	1
1	0	0	1

- A left or right shift for an integer by N bits is equivalent to multiplication or division by 2^N, respectively.
- Shift operators (<< and >>) do not perform an overflow check for plain integers. This means that if the absolute value of the result is more than 2^{31}, the operation deletes extra bits and flips the sign.

Let's discuss a few examples of using the numbers 25 (11001), 50 (110010), and 38 (100110).

```
>>> ~38
-39
>>> ~-38
37
```

Note that bitwise `NOT` for 38 returns -39=-(38+1) and bitwise `NOT` for -38 returns 37=-(-38+1).

```
>>> 38&50
34
>>> 38|25
63
>>> 38^50
20
```

Note that bitwise `AND`, `OR`, and `XOR` compare each bit of the binary values of the operands and evaluate the resulting binary to its decimal value to display the result.

```
>>> 25>>2
6
>>> 38>>3
4
>>> 50<<4
800
```

Observe that a right shift of 25 by 2 is equal to $25/2^2$, which, in turn, is equal to 6. Similarly, a right shift of 38 by 3 is equal to $38/2^3$. The left shift of 50 by 4 is equal to $50*2^4$, which, in turn, is equal to 800.

Membership Operators

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. They are shown in Table 4.4.

Table 4.4 Membership Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`in`	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	`x in y`	`in` results in a `1` if `x` is a member of sequence `y`.
`not in`	Evaluates to false if it finds a variable in the specified sequence and true otherwise.	`x not in y`	`not in` results in a `1` if `x` is a member of sequence `y`.

Let's see a few examples using membership operators.

```
>>> 'a' in 'This is a good software'
1
>>> 12 in ('aaa',12,'abc')
1
```

In the preceding examples, the left operand is searched in the right operand, which is the object of a sequence, and the result is displayed depending on whether the left operand is found in the right operand. Note that the operand on the left of a membership operator can be only a single item.

Identity Operators

Identity operators compare the memory locations of two objects. Table 4.5 lists the identity operators available in Python.

Table 4.5 Identity Operators

OPERATOR	DESCRIPTION	EXAMPLE	EXPLANATION
`is`	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	`x is y`	`is` results in `1` if `id(x)` equals `id(y)`.
`is not`	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	`x is not y`	`is not` results in `1` if `id(x)` is not equal to `id(y)`.

Here are a few examples using the membership operators.

```
>>> p='hello'
>>> ps=p
>>> ps is p
1
```

In the preceding example, the object `'hello'` is stored in the variable `p` and then the variable `p` is assigned to `ps`. This means that both `p` and `ps` contain the same object. Therefore, the `is` operator returns `1`.

```
>>> g=123
>>> f=123
>>> g is f
0
```

Assigning the same values to different variables does not mean that the variables contain the same object. The `is` operator compares the identity of objects. Therefore, the operator returns `0`. Note that objects are considered the same only when they reside in the same memory location. Assigning the same value to another variable does not make the objects equal.

A long expression in a condition may contain multiple operators. In this situation, Python uses the order of precedence to resolve the expression.

Order of Precedence of Operators

Whenever multiple operators are used in a single expression, they are evaluated in a specific order of precedence. To change the way an expression is evaluated, though, you can enclose the expression to be evaluated first in parentheses `()`. Table 4.6 shows the order of precedence of the operators in Python. Those with the same level of precedence are listed in the same row. The order can be changed using parentheses at appropriate places. In all operator groups, the order of precedence is from left to right.

Table 4.6 Order of Precedence of Operators

TYPE	OPERATORS
Boolean logical	`or` `and` `not`
Membership	`in` `not in`
Identity	`is` `is not`
Comparison	`<` `<=` `>=` `>` `<>` `!=` `==`
Bitwise	`|` `^` `&`
Shifts	`<<` `>>`
Additive	`+` `-`

TYPE	OPERATORS
Multiplicative	`*  /  %`
Positive, negative	`+x  -x`
Exponentiation	`**`

After discussing conditional operators, let's discuss how you can use them in programming constructs in the Techsity University scenario where programming constructs are used extensively to create reports of students.

Using Programming Constructs

Problem Statement

Techsity University currently has 50 students and 3 trainers. The university does not have an online site. It has a manual system of creating reports for students. John, a trainer, has been assigned the task of creating reports for all students. He finds it tedious first to manually enter data and then to calculate the total score and grades for 50 students. He is planning to shift to an automated system of generating the reports. He needs to write a program in Python that should first accept the name and registration number of each student and then accept scores for four subjects. The score in any subject cannot be greater than 100. The code written should also check that the user enters all values. The program should calculate the total score and percentage for each student and then evaluate that student's grades based on the following criteria:

SCORE	GRADES
Greater than or equal to 80	A
From 60 through 79	B
From 40 through 59	C
Below 40	Fail

At the end of the calculation for each student, the program should print a report containing the registration number, name, and grade of the student. It should then confirm whether the user wants to continue entering scores for the next student.

Task List

- **Identify the control and loop statements to be used.**
- **Write the code.**
- **Execute the code.**

Identify the Control and Loop Statements to Be Used

Programming constructs are of two types:

- Conditional constructs
- Looping constructs

Let's look at each of these in detail.

Conditional Constructs

Conditional constructs are used to incorporate decision making into programs. The result of this decision making determines the sequence in which a program will execute instructions. You can control the flow of a program by using conditional constructs. These constructs allow the selective execution of statements depending on the value of the expressions associated with them. This section will discuss the programming constructs available in Python, such as `if`, `if...else`, `elif`, and nested `if`.

The `if` Statement

The `if` statement of Python is similar to that of other languages. The `if` statement contains a logical expression using which data is compared, and a decision is made based on the result of the comparison. The syntax of the `if` statement is:

```
if condition:
   statement_true
```

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

In the `if` statement, `condition` is evaluated first. If `condition` is true—that is, if its value is nonzero—then the statements in the `statement_true` block are executed. Otherwise, the next statement following the `statement_true` block is executed. For example,

```
>>> x=10
>>> if x>0:
...   print 'Hello'
```

In this example, the `if` statement prints `Hello` if the value of the variable `x` is greater than `0`.

The `else` Statement

An `else` statement can be combined with an `if` statement. An `else` statement contains the block of code that executes if the conditional expression in the `if` statement resolves to `0` or a false value. The syntax for the `if...else` construct is this:

```
if condition:
   statement_true
else:
   statement_false
```

To print whether a given integer variable is even or odd, the code will be this:

```
if num%2==0:
   print 'even'
else:
   print 'odd'
```

In the preceding example, `'even'` is printed if the modulo of the division of `num` by `2` is `0;` otherwise, `'odd'` is printed.

The `elif` Statement

While making decisions, you may want to include a condition in the `else` statement so that the statements in the `else` block are executed only when that condition is true. The `elif` statement in Python allows you to test multiple expressions for one truth value and executes a particular block of code as soon as one of the conditions evaluates to true. The syntax for an `elif` construct is this:

```
if condition1:
   statement1_true
elif condition2:
   statements2_true
      :
      :
elif condition:
   statementN_true
else:
   statement_none_of_above
```

The following example uses an `elif` construct to determine if the input character is a vowel. Otherwise, it prints an appropriate message.

```
in_chr=raw_input("Enter a character:")
if(in_chr == 'a'):
   print"Vowel a"
elif (in_chr == 'e'):
   print"Vowel e"
elif (in_chr == 'i'):
   print"Vowel i"
elif (in_chr == 'o'):
   print"Vowel o"
elif(in_chr == 'u'):
   print"Vowel u"
else:
   print "The character is not a vowel"
```

The preceding code takes input from the user in the variable `in_chr`. It checks if the value of `in_chr` equals `a`. If the condition is satisfied, it prints `Vowel a`. If the condition is not satisfied, the control of the program moves to the first `elif` statement. This goes on until a satisfying condition is found in the `elif` statements. If a match is found in an `elif` statement, the statement block in that `elif` statement is executed and then

the control comes out of the `if` construct. If it does not find a match, the control moves to the `else` statement. If the user enters the value `o`, the output is `Vowel o`. If the user enters `h`, none of the conditions in `if` and `elif` statements is met. Therefore, the output `The character is not a vowel` is displayed.

Nested `if`

You may also want to see what happens when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested `if` construct. In a nested `if` construct, you can have an `if...else` construct inside another `if...else` construct. The following code uses a nested `if` construct to check if the input character is an uppercase or a lowercase letter.

```
inp=raw_input('Enter a character')
# Find if the character is lowercase or uppercase
if (inp >= 'A'):
    if(inp <= 'Z'):
        print "Uppercase"
    elif (inp >= 'a'):
        if(inp <= 'z'):
            print "Lowercase"
        else:
            print "Input character > z"
    else:
        print "Input character > z but less than a"
else:
    print "Input character less than A"
```

You might be confused by now about deciding which `else` statement belongs to which `if` statement. The answer to this is indentation. Notice that each line in a basic block is indented by the same number of character spaces.

In the preceding example, consider a case where the user enters the value `D`. The execution of the code will be as follows:

1. Check if the input value (D)>= 'A'. If yes, check if the input value <= 'Z'.
2. In this case, because the input value (D) >= 'A' and <= 'Z', the output will be `Uppercase`.

Similarly, consider the case where the user enters `f`. The execution of the code would be as follows:

1. Check if the input value (f)>= 'A'. Because the input value `'f'` is not >= 'A' and is not <= 'Z', the control passes to the `elif` statement that belongs to the second `if` statement.
2. The `elif` statement will check if the input value >= 'a' and <='z'. Because it is, the output will be Lowercase.

If all the preceding three `if` statements return false, the output will be `'Input character > z'`.

So far, all the conditional statements that you have worked with execute in a linear fashion. This means that these statements execute only once and stop when the condition is fulfilled. There might be situations, though, when you want a set of statements to repeat a specific number of times. Loops can be used to achieve this type of functionality.

Looping Constructs

A *loop* is a construct that causes a section of a program to be repeated a certain number of times. The repetition continues while the condition set for the loop remains true. When the condition becomes false, the loop ends and the program control is passed to the statement following the loop. As against conditional constructs, which execute only once, loop constructs execute continuously until the evaluating condition is no longer satisfied. This section will discuss the various looping constructs in Python, such as `while` and `for` statements. In addition, you will also learn about `break` and `continue` statements, which are used with looping constructs.

The `while` Loop

The `while` loop is one of the looping constructs available in Python. The `while` loop continues until the evaluating condition becomes false. The evaluating condition has to be a logical expression and must return either a true or a false value. The general syntax for the `while` loop is this:

```
while evaluating_condition:
   repeat_statements
```

Figure 4.1 shows the sequence of steps for the execution of a `while` loop.

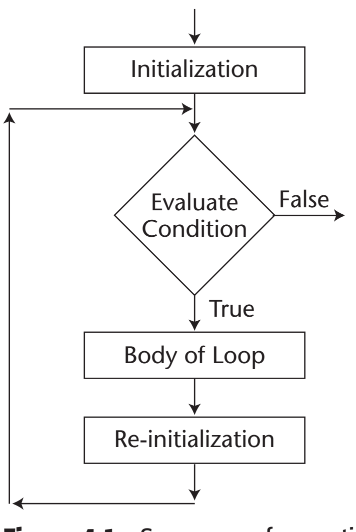


Figure 4.1 Sequence of execution of the `while` loop.

The following code demonstrates the use of the `while` loop:

```
n=input('Enter a number greater than 1  ')
num1=0
num2=1
print num1
while(num2<n):
   print num2
   num2=num1+num2
   num1=num2-num1
```

The output of the preceding code when the user enters the value `200` will be:

```
enter a number greater than 1  200
0
1
1
2
3
5
8
13
21
34
55
89
144
```

In the preceding code, the statements indented in the `while` loop are executed repeatedly until the variable `num2` becomes greater than `200`. With each iteration, the value of `num2` increments by the value of `num1`, and the value of `num1` changes to the current value of `num2` after subtracting `num1`.

Infinite Loops

You must be extremely cautious while using `while` loops because the condition you specify in a `while` loop might never resolve to a false value. This results in a loop that never ends. Such a loop is called an infinite loop. An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required. Infinite loops, however, do not generate the required output in all types of programs. The execution of an infinite loop depends on whether the loop was supposed to run forever. Consider the following code snippet:

```
>>>i=1
>>>while i==1:
...   reg_no=raw_input("Please enter your reg number    ")
...   tot_score=score(reg_no)
...   print "Your total score is  ", tot_score
```

In the preceding code, the value of the variable `i` is initialized to `1`. The first statement inside the `while` loop asks for user input. The second statement calls the

score() function that evaluates the total score. The third statement prints the total score evaluated by the score() function. Therefore, the while loop executes until the value of i remains 1. This loop will execute infinitely because the condition in the while loop will never resolve to Boolean false. Let's change the preceding code to solve the problem as shown here:

```
>>>i=1
>>>while i==1:
...   reg_no=raw_input("Please enter your reg number    ")
...   if valid(reg_no)==1:
...      tot_score=score(reg_no)
...      print "Your total score is  ", tot_score
...      i=0
...   else:
...      print "You haven't entered a valid registration number"
```

Now, you have successfully solved the problem of an infinite loop. In this code snippet, if a user enters a valid registration number, the total score of the user is calculated and the variable used in the condition is assigned to Boolean false. The loop is executed again only when the user enters an invalid registration number.

The break Statement

You might face a situation in which you need to exit a loop when an external condition is triggered. What do you do in such a situation? The break statement comes to your rescue. It causes the program flow to exit the body of the while loop and resume the execution of the program at the next statement after the while loop. The break statement can be used to force an early exit from a loop or to implement a loop with a test to exit in the middle of the loop body. A break statement within a loop should always be protected within an if statement that enables you to check the exit condition.

```
a=input('Enter an integer ')
num1=0
num2=1
print num1
while (num2 < a):
   print num2
   num2 = num1+num2
   num1 = num2 - num1
   if num2==89
      break
```

The preceding code is the same code that generates the Fibonacci series. The if statement inside the while loop checks if the value of num2 is 89 and uses the break statement to terminate the loop. The output of the code will be this:

```
Enter an integer 200
0
1
1
2
```

```
3
5
8
13
21
34
55
```

The `continue` Statement

The `continue` statement returns the control to the beginning of the `while` loop. There is, however, a misconception among budding programmers that the `continue` statement continues to execute the current iteration of the loop and moves the control to the next iteration. This statement is correct to a certain extent. What does the `continue` statement actually do? Instead of continuing the current iteration and then returning the control to the beginning of the loop, the `continue` statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. Therefore, it skips any statement following the `continue` statement in the body of the loop. The following code illustrates the `continue` statement:

```
num=0
reply='y'
while(reply!='n'):
    num=int(raw_input("Enter a number: "))
    if(num>100):
        print "The number is greater than 100. Enter again"
        continue      # The statements after this will be skipped
    print "The square of the number is ",num*num
    reply=raw_input("Do you want to enter another y/n  ")
```

This code requests a user to enter a value and prints its square. If the value entered by the user is greater than 100, the user is asked to enter a value again. This `while` loop is executed as many times as the user wants to input a value. Save this code as a `.py` file. Run the module on the Shell prompt. Figure 4.2 shows a sample output of the code.

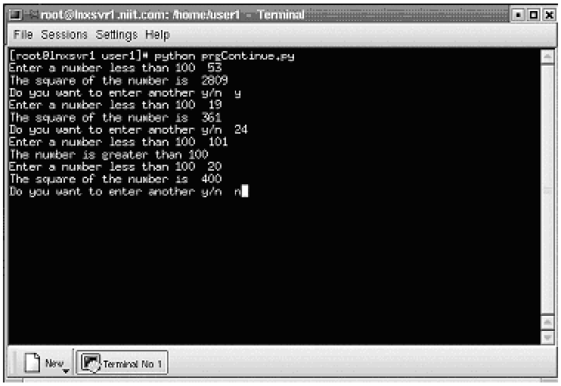


Figure 4.2 Output of the code showing the `continue` statement.

The `for` Loop

The `for` loop in Python has the ability to iterate over the items of any sequence, such as a list or a string. The general syntax of the `for` loop is this:

```
for iterating_var in sequence:
   statements_to_repeat
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable. Next, the statements in the `statements_to_repeat` block are executed. Each item in the list is assigned to `iterating_var`, and the `statements_to_repeat` block is executed until the entire sequence is exhausted. Let's see how the `for` loop works with different types of sequences. Consider the following example for a string sequence.

```
>>> for letter in 'Greece':
...   print 'current letter:', letter
```

The output of the preceding code will be this:

```
current letter: G
current letter: r
current letter: e
current letter: e
current letter: c
current letter: e
```

When a `for` loop iterates over a string, the iteration variable assumes the value of a single character in each iteration. You might not use such a `for` loop to traverse a string. A more useful way to traverse a string is to use the `in` operator or one of the module functions that we covered in Chapter 2, "Getting Stated with Python." The output of a `for` loop that is printed in the form of single characters can be used to indicate that a string is used as the sequence in the `for` loop rather than a sequence of objects, such as a list.

You might want to print a list containing the names of people along with the lengths of their names. A `for` loop can be easily used for this purpose. Let's see how:

```
>>>names=['Laurie','James','Mark','John','William']
>>> for x in names:
...    print 'The name %-3s is %d characters long' % (x,len(x))
```

In this code, the `%` symbol is used to format the output of the code and the `len()` function is used to calculate the length of each item in the list `names` . The output of this code will be this:

```
The name Laurie is 6 characters long
The name James is 5 characters long
The name Mark is 4 characters long
The name John is 4 characters long
The name William is 7 characters long
```

We have seen how to use the `for` loop when we have to iterate over a string or a list. But how do we iterate over a sequence of numbers? Let's see how.

The `break` and `continue` Statements

In an earlier section of this chapter, you learned how to use `break` and `continue` statements in a `while` loop. You can also use the `break` and `continue` statements in a `for` loop. The `break` statement breaks out of the current iteration of a loop typically when a condition is met. For example,

```
>>> names=['Laurie','James','Mark','John','William']
>>> for i in range(len(names)):
...   if len(i)>6:
...      break
...      print i,names[i]
...
0 Laurie
1 James
2 Mark
3 John
```

The preceding code iterates over list `names` and finds an item in the list whose length is greater than 6. As soon as this happens, the `for` loop breaks.

You can use the `continue` statement to continue with the next iteration of the loop. Consider the following code in which the user is asked to enter a password and a maximum of three chances is allowed. If the password is found valid, then the current iteration of the `for` loop is interrupted. Otherwise, the iterating variable is decremented by 1 and the loop is continued to the next iteration.

```
valid=0
i=3
while i>0:
   inp=raw_input("Enter the password")
   for pass in passwordlist:
      if inp==pass:
      valid=1
      break
   if not valid:
      print "invalid password"
      i=i-1
      continue
   else:
      break
```

The `else` Statement Used with Loops

In Python, a loop can also have an `else` statement associated with it. How does this work? If the `else` statement is used with a `for` loop, the `else` statement is executed when the loop has exhausted iterating the list. The following example illustrates the combination of an `else` statement with a `for` statement that searches for prime numbers from 10 through 20.

```
>>>for num in range(10,20):   #to iterate on every number
...#between 10 to 20
...   for i in range(2,num):   #to iterate on the factors of the
...   #number
...      if num%i==0:   #to determine the first factor
...         j=num/i   #to calculate the second factor
...         print '%d equals %d * %d' % (num,i,j)
...         break      #to move to the next number, the
...               #first FOR
...   else:
...      print num, 'is a prime number'
```

The output of the preceding code will be this:

```
10 equals 2*5
11 is a prime number
12 equals 2*6
13 is a prime number
14 equals 2*7
15 equals 3*5
16 equals 2*8
17 is a prime number
18 equals 2*9
19 is a prime number
```

The `pass` Statement

The `pass` statement is used when a statement is required syntactically but you do not want any command or code to execute. The `pass` statement is a null operation; nothing happens when it executes. For example,

```
>>> a=1
>>> if a==1:
         pass
```

The preceding code does not execute any statement or code if the value of `a` is `1`. The `pass` statement is helpful when you have created a code block but it is no longer required. You can then remove the statements inside the block but let the block remain with a `pass` statement so that it doesn't interfere with other parts of the code.

Result

Based on the preceding discussion, the programming constructs that you will use to generate reports are as follows:

- The report has to be generated for a maximum of 50 students. Therefore, you need to have a `while` loop that executes until the number of students reaches 50.
- You need to check whether the user has not skipped entering the name, registration number, and score in any subject. You can do this by using an `if` or a

while statement. Using the while statement with a variable having a Boolean true value will be the appropriate option. This variable should become false when the user enters a value for a name or a registration number; the while loop will repeat itself until the user does not enter a value for the variable. If you use the if statement, you cannot force the control of the program to go back to the statement that checks if the user does not enter a value again.

- You also need to have if and elif statements to decide the grade of a student based on the specified criteria.
- Then, you need to have a statement that checks whether the user wants to continue entering details for another iteration of the first while loop. You can use an if...else construct for this purpose. This if statement will check if the user input matches the allowed sequence of items in a list.
- This if statement should also contain a statement to increment the value of the counter variable used in the first while loop. The else statement should contain a break statement to end the first while loop if the user does not want to continue.

Write the Code

The code for the problem statement in the beginning of the chapter is as follows:

```
i=1
while (i<=50): #Repeats the loop 50 times
    while 1:
        #Takes user input for name and then checks if its length is 0
        name=raw_input("Enter the name ")
        if len(name)==0:
            continue
        else:
            break
    while 1:
        #Takes user input for name and then checks if its
        #length is not 0
        reg_no=raw_input("Enter the registration number ")
        if len(reg_no)==0:
            continue
        else:
            break
    j=1
    tot_score=0
    while j<=4:
        #Iterates 4 times for 4 subjects
        print 'Enter score in subject', j
        score=raw_input()
        if len(score)==0 or int(score)>100:
            #Checks for the length of score and that they are
            #not greater than 100
            continue
        else:
```

```
            j=j+1
            #Goes back to next iteration only when user has
            #entered a valid value
            tot_score=tot_score+ int(score) #Calculates total score
    percent=tot_score/4
    if percent>=80:
        #Evaluates  the grade
        grade='A'
    elif percent>=60:
        grade='B'
    elif percent>=40:
        grade='C'
    else:
        grade='Fail'
    for clear in range(35):
        #Prints 40 blank lines
        print
    print '-'*60
    print "Name             :", name
    print
    print "Registration no. :", reg_no
    print
    print "Grade            :", grade
    print '-'*60
    for y in range(10):
        print
    choice=raw_input("Do you want to enter details for \
         another student?  ")
    choices=['y','Y','yes','Yes','YES']
    if choice in choices:
        #Checks if the user wants to add another value
        i=i+1
        for clear in range(40):
            #Prints 40 blank lines
            print
    else:
        break
```

Execute the Code

To be able to view the output of the preceding code, execute the following steps:

1. Type the code in a text editor.
2. Save the file as **prgStat.py**.
3. Make the directory where you have saved the file the current directory.
4. On the Shell prompt, type:

   ```
   python prgStat.py.
   ```

Figure 4.3 shows the sample input.

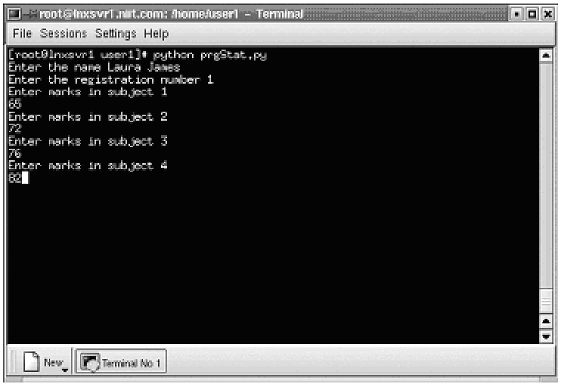


Figure 4.3 Sample input for executing the code.

Figure 4.4 shows a sample output.

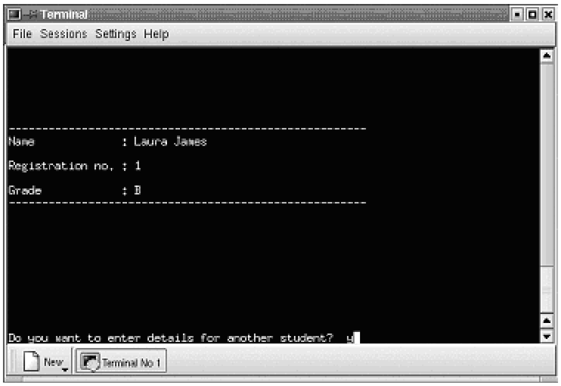


Figure 4.4 Sample output.

Summary

In this chapter, you learned that:

- Conditional operators are used to compare values and test multiple conditions. They are classified as:
  - Comparison operators
  - Bitwise operators
  - Boolean logical operators
  - Membership operators
- Conditional constructs are used to allow the selective execution of statements. The conditional constructs in Python are these:

  ```
  if...else
  elif
  nested if
  ```
- Looping constructs are used when you want a section of a program to be repeated a certain number of times. Python offers the following looping constructs:

  ```
  while
  for
  ```
- The `break` and `continue` statements are used to control the program flow within a loop.

CHAPTER

5

Functions

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Use user-defined functions**
- **Use the following arguments:**
  - **Formal:**
    - **Required**
    - **Keyword**
    - **Default**
  - **Variable-length:**
    - **Keyword**
    - **Non-keyword**
- **Pass functions as arguments**
- **Return values from functions**
- **Use lambda forms**

Continues

OBJECTIVES (CONTINUED)

- **Use built-in functions:**
  - `apply()`
  - `filter()`
  - `map()`
- **Scope variables**

Getting Started

In Chapter 4, you learned about programming constructs, which are used to make choices or perform certain actions based on whether a particular condition is true. In a real-life scenario, your code will be written in multiple code blocks that perform multiple actions. When these code blocks are written one after the other, the readability of the code is affected. Another programmer reading the code may not understand the action performed by each code block. Moreover, in order to reuse a specific code block, you need to create multiple copies. This chapter will move a step further and discuss functions.

Functions add reusability, modularity, and overall programming simplicity to code. Functions also provide programmers with a convenient way of designing programs in which complex computations can be built. After a function is properly designed, a programmer does not need to bother about how the calculations are done in it. It is sufficient that the programmer knows what the function does and simply ensures that the required parameters are passed to it. Therefore, to the programmer, the function itself becomes a black box.

Consider a situation in which the execution of a lengthy program generates an error. It is difficult to debug such a program manually by scanning each line of code. You can break up the related lines of code into functions. This helps you debug only the piece of code that does not execute properly or generates errors.

Using Functions

Problem Statement

Techsity University wants to add a sign-up page on its Web site for its students. The sign-up page will accept user details, such as first name, last name, and date of birth. Based on these details, the page should suggest four possible login ids. The criteria for suggested login ids are stated in the text that follows. The examples provided here are based on the name John Smith and the date of birth December 24, 1978.

Login1. First name followed by the first letter of the last name (for example, johns).

Login2. First name followed by the first letter of the last name and the month and day of birth (for example, johns1024).

Login3. First letter of the last name followed by first name and the year of birth (for example, sjohn78).

Login4. First letter of first name followed by the last name and the age (for example, jsmith23).

Task List

- ✔ **Identify the functions to be used.**
- ✔ **Write the code.**
- ✔ **Execute the code.**

Let's learn about functions that help us solve this problem.

Functions

A *function* is a block of organized, reusable code that is used to perform a single, related action. In other words, it is a set of related statements that provides a structured way of organizing the logic in your program. Python supports the following types of functions:

- User-defined functions
- Lambda forms
- Built-in functions

User-Defined Functions

You can define functions to provide the required functionality. Function blocks begin with the keyword `def` followed by the function name and parentheses (**()**). Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses. The first statement of a function can be an optional statement—the documentation string of the function or *docstring*. The code block within every function starts with a colon (:) and is indented. The syntax for a function declaration is this:

```
def functionname(parameters):
    "function_docstring"
    function_suite
```

The following example illustrates a user-defined function:

```
def fnsquare(num):
    x=0
    x=num*num
    print x
```

In the preceding example, the name of the function is `fnsquare` and the block of code consists of the statements that are indented after the colon.

After you have defined a function, you need to execute it to implement its functionality.

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function, and structures the blocks of code. After the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. You can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. In addition, the number of arguments in the function call should match exactly with the function definition. To call the function `fnsquare` from the Python prompt, execute the following statement:

```
>>>fnsquare(40)
```

This function call will pass the value `40` to `num` in the function definition. The function will, in turn, execute the statements inside the function body assuming the value of `num` to be `40`. The output of this function call will be `1600`.

The function call should have exactly the same number of arguments that are defined for the function that is called. In addition, the order in which the arguments are placed in the function call should be the same as the order in which they were defined. For example, the function `fnsquare` is defined for one argument only. Therefore, it can take only one input value. If the function is called in any of the following ways, it will return an error:

```
>>> fnsquare()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in ?
TypeError: fnsquare() takes exactly 1 argument (0 given)
>>> fnsquare(15,'hi')
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in ?
TypeError: fnsquare() takes exactly 1 argument (2 given)
>>> fnsquare(3.2)
10.24
```

Python, however, allows you to change the order of arguments or skip them. This can be done using keyword arguments.

Keyword Arguments

Keyword arguments are related to the function calls. When you use *keyword arguments* in a function call, the caller identifies the arguments by the parameter name. Therefore, this allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. Let's consider the following example of the function to illustrate keyword arguments:

```
>>>def printx(x):
...   print x
```

The standard calls to this function will be:

```
>>>printx('fff')
>>>printx(32)
>>>y='abc'
>>>printx(y)
```

You can also make keyword calls to the function in the following ways:

```
>>>printf(x='fff')
>>>printx(x=32)
>>>y='abc'
>>>printx(x=y)
```

The following function illustrates a more realistic example. Consider a function, `stud_fn()`.

```
>>>def stud_fn(reg_no, name, score):
...   print 'The score for the student, ', reg_no, ' , is', score
```

You can use keyword arguments to call this function in any of the following ways:

```
>>>stud_fn(score=86,reg_no='S001', name='Laura')
```

In this function call, the `score` parameter assumes the value `86`, `reg_no` assumes the value `S001`, and `name` assumes the value `Laura`. Notice that the order of parameters is different in the definition of the function and its call. Keyword arguments allow out-of-order arguments, but you must provide the name of the parameter as the keyword to match the values of arguments with their corresponding names.

You can also skip arguments by using keyword arguments, as in the following:

```
>>>stud_fn(score=86,reg_no='S001')
```

In the preceding function call, you have used the keyword argument to "miss" the argument `name`. This is allowed, though, only when you are using default arguments.

Default Arguments

When your functions have many arguments, it becomes a tedious job to pass values for each of them. In such cases, you can specify default arguments. A *default argument* is an

argument that assumes a default value if a value is not provided in the function call for that argument. This is extremely helpful for a programmer who has to extend the code written by another programmer and does not have adequate knowledge to provide more values as arguments. Another advantage of using default arguments occurs while developing an application for an end user. When you provide default values, you provide the consumer with the freedom of not having to choose a particular value. The following example illustrates a situation in which a default argument is useful for a Web scenario:

```
>>>def course_fee(fee, discount=0.15):
...     print fee-(fee*discount)
...
>>>course_fee(500)
425.0
```

In the preceding code, the `course_fee` function takes the fee for a course as a parameter and then displays the fee after subtracting the discount. In this code, `fee` is the required parameter and `discount` is the default parameter, which takes a default value of 15 percent. Students taking different courses in the university would want to see `course_fee` that they have to pay for a course or a semester. If a student shows extraordinary performance, then the university may want to give the student a higher discount. You can override the default value of the discount by providing a different value for the default argument. In the preceding example, you can specify a discount of 20 percent while calling the `course_fee` function.

```
>>>course_fee(500,0.20)
400.0
```

By specifying a value for `discount`, you override the default value of `0.15`, which was specified when the function was defined. There is one thing that you should keep in mind while specifying both default and required parameters for the same function: You must place all the required parameters before the default parameters in the function definition. While calling the function, default parameters do not have to be necessarily specified. Therefore, if mixed modes were allowed, it will become very difficult for the interpreter to match each value with its corresponding parameter. A syntax error is raised if the order of the parameters is not correct.

```
>>>def course_fee(discount=0.15,fee):
...     print fee-(fee*discount)
SyntaxError: non-default argument follows default argument
```

You can change the order of default and nondefault arguments by using the keyword arguments in the function call. Let's look at keyword arguments again in relation to default arguments. Consider the following code snippet that calculates the area of a shape.

```
>>>def shape(type,radius=3,height=4,length=5):
...     suite
```

Table 5.1 Invalid Function Calls to the `shape()` Function

FUNCTION CALL	EXPLANATION
`shape( )`	Required argument not specified.
`shape('circle',type='cone')`	Duplicate value of the parameter specified.
`shape(radius=3,'sphere')`	Default argument specified before a nondefault argument.
`shape(type='sphere',3)`	Keyword argument specified before a non-keyword argument.
`shape(perimeter=30)`	Unknown keyword specified.

This function can be called in any of the following ways:

```
>>>shape('circle')
>>>shape(radius=12,type='sphere')
>>>shape('cone',3,4,5)
>>>shape(cylinder,3,4)
```

Table 5.1 lists the calls to the `shape()` function that will be invalid.

While calling a function, an argument list must first contain positional arguments followed by any keyword argument. Keyword arguments should be taken from the required arguments only. Moreover, you cannot specify a value for an argument more than once.

Variable-Length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length arguments* and are not named in the function definition, unlike required and default arguments. You can use these arguments when the number of arguments is unknown before run time or when the number of arguments is different for subsequent calls of the function. Python supports both keyword and non-keyword variable arguments.

Non-keyword Variable Arguments

When you call a function, all formal arguments are assigned to their corresponding variables as specified in the function declaration. The remaining *non-keyword* variable arguments are assigned to a tuple. Variable-length arguments should follow all formal parameters. The general syntax for a function with non-keyword variable arguments is this:

```
def function_name([formal_args,] *var_args_tuple)
    suite
```

An asterisk (*) is placed before the variable name that will hold the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Let's consider an example to demonstrate the use of non-keyword variable arguments.

```
>>> def tuple_func(formal1, formal2='xyz', *vartuple):
...     print 'formal argument 1 ',formal1
...     print 'formal argument 2 ',formal2
...     for each in vartuple:
...         print ' another argument ',each
```

Let's call this function with different values.

```
>>>tuple_func('city','state', 20)
```

The output of this call will be:

```
formal argument 1 city
formal argument 2 state
another argument 20
```

The first two values in the function call are assigned to the formal arguments `formal1` and `formal2`. There are no more formal arguments. Therefore, the tuple `vartuple` assumes the third value.

```
>>>tuple_func('city')
```

The output of this call will be:

```
formal argument 1 city
formal argument 2 xyz
```

The value in the function call is assigned to the first nondefault formal argument, `formal1`. The default argument, `formal2`, contains the value that was assigned to it during the function declaration. There are no further arguments present in the function call; therefore, the tuple `vartuple` remains empty.

```
>>>tuple_func('USA','state',20,'New York',30.2)
```

The output of this call will be:

```
formal argument 1 USA
formal argument 2 state
another argument 20
another argument New York
another argument 30.2
```

The first two values in the function call are assigned to the formal arguments `formal1` and `formal2`. There are no more formal arguments left; therefore, the rest of the values go into the tuple `vartuple`.

Keyword Variable Arguments

You have already learned that when you call a function, first all formal arguments are assigned to their corresponding variables. Then, the remaining non-keyword variable arguments are assigned to a tuple. If there are still any more keyword arguments after this, they are assigned to a dictionary. The general syntax for a function with the keyword variable arguments is this:

```
def function_name([formal_args,][*var_args_tuple,] **var_args_dict)
    suite
```

A double asterisk (**) is placed before the variable name that holds the values of all keyword variable arguments. Let's consider an example to demonstrate the use of non-keyword variable arguments.

```
>>> def tuple_func(formal1, formal2='xyz',**vardict):
...    print 'formal argument 1 ',formal1
...    print 'formal argument 2 ',formal2
...    for each in vardict:
...       print 'another keyword argument %s=%s' %(each,vardict[each])
```

Let's call the preceding function:

```
>>>tuple_func('city','state', num=20.2,count=30)
```

The output of this call will be:

```
formal argument 1 city
formal argument 2 state
another keyword argument count=30
another keyword argument num=20.2
```

The first two values in the function call are assigned to the formal arguments `formal1` and `formal2`. There are no more formal arguments or non-keyword variable arguments. Therefore, the third keyword value becomes a part of the dictionary `vardict`. Similarly, the fourth keyword value becomes a part of `vardict`.

You can use both keyword and non-keyword variable arguments in the same function. In such a case, however, the keyword variable arguments should follow the non-keyword variable arguments.

```
def var_args_func(formal1, formal2='xyz',*vark,**varnk):
    print "formal argument 1 ",formal1
    print "formal argument 2 ",formal2
    for eachk in vark:
        print'another keyword argument: ', eachk
    for eachnk in varnk:
        print'another non-keyword argument %s=%s' \
%(eachnk,str(varnk[eachnk]))
```

When you call var_args_func within the interpreter, the following output is obtained:

```
>>> var_args_func('city',30,'USA','2000',reg=30,que=42)
formal argument 1  city
formal argument 2  30
another keyword argument:  USA
another keyword argument:  2000
another non-keyword argument que=42
another non-keyword argument reg=30
```

The `return` Statement

You have learned that you can use arguments to pass input values to functions. You can also make a function return a value by using the return statement. Returning a value means that when the function is called, it returns the result of its operation to a variable. The following example illustrates the use of the return statement.

```
>>>def add(num1,num2):
...    return num1+num2
```

The preceding function takes two input parameters, adds them, and returns the result.

The return value can be accessed in the following way:

```
>>>sum=add(14,24)
>>>print sum
```

In the preceding lines, the value returned by the add() function is stored in the variable sum. This value can be retrieved at the Python prompt by using the print statement. Values returned by a function can be passed to another as a parameter, as in the following code, which calculates the square of the value returned by the add() function.

```
>>>def square(sum):
...    return sum*sum
```

The entire code that implements both of these functions will be:

```
def main_func(a,b):
    def add(num1,num2):
        return num1+num2
    sum=add(a,b)    #Return value of add stored in sum
    def square(sum):     #calculates the square of sum
        return sum*sum
    sq=square(sum)
    return sq
```

If the function call is made at the interpreter, the output of the preceding code will be:

```
>>>output=main_func(14,24)
>>>output
1444
```

When a call is made to the `main_func` function with input values `14` and `24`, they are stored in variables, `a` and `b`. The preceding code calculates the sum of two numbers, `num1` and `num2`, in the `add()` function and stores the return value in `sum`. The values of `a` and `b`, which are `14` and `24`, are passed to `add()`. The values are then stored in `num1` and `num2`. The `square()` function takes `sum` as the input parameter and returns the square of `sum`. The variable `sq` in `main_func` is used to store the return value of the square function. The `main_func` then returns the value of `sq`.

NOTE Python does not allow you to call a function before the function is declared.

Python allows you to return only one value or object from a function. How do you return multiple values by using a `return` statement? You can do this by using a tuple or a list. Consider the following example:

```
>>>def func():
...    return ('fff','city',8023)
```

This function returns a tuple. You can store the values of this tuple in a variable as follows:

```
>>>tup=func()
>>>tup
('fff','city',8023)
```

In this assignment, the variable `tup` stores the entire tuple, which is returned by the `func` function. You can also use as many variables as the number of parameters in the function to assign the values.

```
>>>(a,b,c)=func()
>>>(a,b,c)
('fff','city',8023)
```

or

```
>>>p,q,r=func()
>>>p,q,r
('fff','city',8023)
```

In the preceding assignments, each of the variables `a`, `b`, `c` and `p`, `q`, `r` will be assigned to its corresponding return value.

Passing Functions

In Python, functions are treated like any other object. The value of the function name has a type. The interpreter recognizes this type as a user-defined function. This value can be assigned to another variable, which can then be used as a function itself. It can also be passed to other functions as arguments or can be elements of other objects, such as lists and dictionaries.

Functions can be aliases to variables, but what are aliases? Let's understand this by using an analogy. A person may be addressed as Robert Jenkins by strangers and Bob by his wife. Both the name and the alias actually refer to the same entity. In the following code, there is a single function called `ruf` with an alias called `bee`.

```
>>>def ruf():
...   print 'ruf----'
...
>>>bee=ruf
>>>bee()
ruf----
```

While assigning `bee` to `ruf`, the same function object was assigned to `bee`. Therefore, `bee()` can be invoked in the same way that `ruf()` was invoked.

NOTE When you write the name of a function without parentheses, it is interpreted as the reference, such as `ruf`. When you write the function name with parentheses, such as `ruf()`, the interpreter invokes the function object.

You can even pass function references to other functions as arguments. Let's discuss this in the following example:

```
>>>def bee(arg):
...   arg()
```

The function can be called from the interpreter by using the following statement:

```
>>>bee(ruf)
ruf----
```

When you write the preceding statement on the Python prompt, the function object `ruf` is passed to `bee` as an argument. The function `bee()` calls `ruf()` by assigning `arg` to `ruf`. When `arg()` is executed in `bee`, `ruf()` is executed. You can also pass objects of built-in functions to other functions as arguments.

Lambda Forms

You can use the lambda keyword to create small *anonymous* functions. These functions are called anonymous because they are not declared in the standard manner by using the `def` keyword. Lambda forms can take any number of arguments but return just

one value in the form of an expression. They cannot contain commands or multiple expressions. Lambda forms can be used whenever function objects are required; however, lambda forms do not create any names in the namespace if they are not assigned to a variable. The entire syntax of lambda functions contains only a single statement, which is as follows:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

A corresponding single statement function will be:

```
def functionname([arg1 [,arg2,.....argn]]):
   expression
```

Let's understand this better by considering the following single statement function:

```
def func():
   return 'Hi'
```

The preceding function does not take any arguments and always returns `'Hi'`. You can write the same code in Python in a single line as:

```
def func():   return 'Hi'
```

You can write this single statement function in the lambda form as follows:

```
lambda: 'Hi'
```

Table 5.2 contains some more examples of functions and their lambda forms.

A call to a lambda function cannot be made directly. It has to be either assigned to another variable or returned as a part of another function. The following examples illustrate calling lambda forms:

```
>>>x= lambda a,b: a*b
>>>x(5,6)
30
```

Table 5.2 Examples of Single Statement Functions and Their Lambda Forms

FUNCTION	LAMBDA FORM
`def prod(a,b) :return a*b`	`lambda a,b: a*b`
`def prod2(a,b=6) :return a*b`	`lambda a,b=6:a*b`
`def tuple_arg(*tup):return tup`	`lambda *tup: tup`
`def many_args(tup=('a','b'),**dt): return [tup,dt]`	`lambda tup=('a','b'),**dt:`

In the preceding statements, the variable x is assigned to the lambda form. When you invoke x with the arguments 5 and 6, the corresponding lambda form is invoked. The values are passed to a and b, the product of a and b is calculated, and the output is displayed.

```
>>>x=lambda a,b=6:a*b
>>>x(10)
60
```

In the preceding statements, b is a default argument; therefore, you do not need to supply a value to b. The lambda form can be invoked by assigning the lambda form to the variable x. The value returned by the lambda expression is 60.

```
>>>s= lambda *tup:tup
>>>s('hgh',23)
('hgh', 23)
```

In the preceding statements, the argument tup of type tuple is passed to the lambda expression. The lambda expression is called using the variable s. Note that the arguments passed to the lambda expression are returned in the form of a tuple.

You can also enable a lambda form to return multiple values in the form of a list, a tuple, or a dictionary. The preceding example used a tuple. Let's examine an example in which the return value is stored in a list.

```
>>>s= lambda tup=('hgh',23),**dt: [tup,dt]
>>> s((56,23),f=656,g=23,h=23)
[(56, 23), {'h': 23, 'g': 23, 'f': 656}]
```

The lambda function mentioned previously takes two arguments, tup and dt. tup is the default argument with two values, 'hgh' and 23, and dt is the keyword variable argument. Note that the return variables are enclosed within square brackets, which means that the return values of both tup and dt will be stored in a list. As in the other examples discussed before, the lambda form is assigned to the variable s. Therefore, this variable will now store the list containing the tuple and the dictionary. The first two values were in enclosed parentheses; therefore, while calling the lambda form, the values are passed to tup. The rest of the keyword values are passed to dt. Notice that the first two values in the output belong to a tuple and the rest to a dictionary.

Built-In Functions

You have already learned about the built-in functions that perform operations on data structures. In this section, you will learn about some more built-in functions that are not related to a specific data structure. This chapter will discuss some of the built-in functions here, such as apply(), filter(), map(), and reduce().

The apply() Function

The apply() function does not do anything different than any other user-defined or built-in function does. When a function and its parameters are passed to the apply()

function as arguments, the apply() function invokes an object of that function. The syntax of the apply() function is this:

```
apply(object[,args[,kwargs]])
```

The function call ruf('sdsd',23) is the same as the function call apply(ruf,('sdsd',23)). You can also pass the arguments of the ruf() function to the apply() function by using a tuple as follows:

```
tup=('sdsd',23)
apply(ruf,tup)
```

Each element of tup is passed to ruf() as a separate argument by using the apply() function instead of an argument, which is tup. Why do you actually need to use the apply() function? After all, it just invokes another function, which can be done directly by calling that function. The apply() function is most useful when the arguments that need to be passed to a function are generated at run time. Consider the following example, which takes two numbers and an operator as user input. Based on the operator, the corresponding math function is applied to the two numbers and the result is displayed.

```
from operator import add, sub, mul
op=('+','-','*')
nums=(int(raw_input('Enter number 1: ')),int(raw_input('Enter number 2:
')))
ops={'+':add,'-':sub,'*':mul}
ch=raw_input('Enter an operator, +/-/*: ')
if ch in op:
    ans=apply(ops[ch],nums)
    print '%s%s%s=%d' %(nums[0],ch,nums[1],ans)
else:
    print 'invalid operator'
```

Let's understand the apply() function by using the preceding code:

- op is a tuple containing all the possible values of operators that are allowed. nums is another tuple that contains the two numbers entered by a user. ops is the dictionary containing the corresponding math function objects that match the operators.
- Using the if construct, the function checks whether the name of the operator specified by the user is in op. The function then invokes apply() to call the math function with the operator and two numbers in order to calculate the correct solution.

The sample output of the code will be:

```
Enter number 1: 8
Enter number 2: 5
Enter an operator, +/-/*: *
8*5=40
```

The `filter()` Function

This function filters the items of a sequence based on a Boolean function and returns another sequence of the same type. The sequence returned by the `filter()` function contains only those values for which the Boolean function returns `true`. The syntax of the `filter()` function is this:

```
filter(boo_func,sequence)
```

You can understand the working of the `filter()` function by reviewing Figure 5.1.

Let's start with the original sequence `seq`, which has the elements `seq[0]`, `seq[1]`, `seq[2]`, ... `seq[N]`. For each item in the list, the Boolean function `boo_func` is called. This function returns either 0 or 1 for each item. The result is another sequence of the same type that contains the items for which `boo_func` has returned the value `true`. The `filter()` function returns only the newly created sequence. Consider the following example, which accepts user input for the number of years, determines the leap years, and returns a sequence containing leap years.

```
def leap(n):
    return n%4==0
list_yr=[]
ch=raw_input('Do you want to enter a year? ')
while 1:
    if ch in ('y','yes','Y'):
        yr=raw_input('Enter a year:')
        list_yr.append(int(yr))
        ch=raw_input('Do you want to enter another year? ')
    else:
        break
leap_yrs=filter(leap,list_yr)
print 'You have entered %d leap years, they are: \
%s'%(len(leap_yrs),leap_yrs)
```

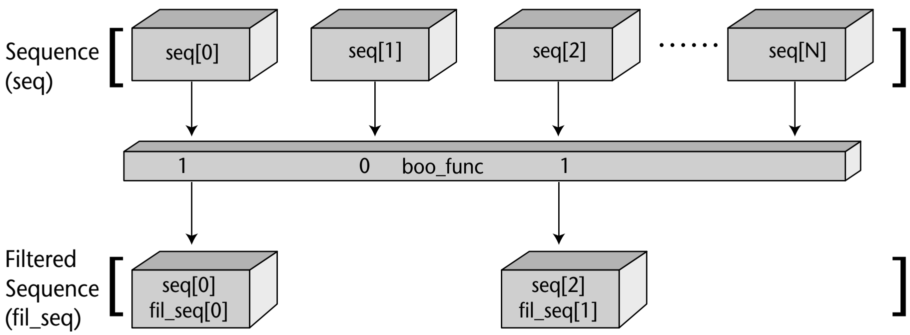


Figure 5.1 Working of the `filter()` function.

By now, you are in a good position to understand this code. The preceding code accepts user input values for the number of years and adds the year to `list_yr`. This happens until the user wants to enter more values. `filter()` is called to determine the leap years in the `list_yr` and stores them in `leap_yrs`. Here is an output sample for this code:

```
Do you want to enter a year? y
Enter a year:2000
Do you want to enter another year? y
Enter a year:1998
Do you want to enter another year? y
Enter a year:1954
Do you want to enter another year? yes
Enter a year:1564
Do you want to enter another year? n
You have entered 2 leap years, they are: [2000, 1564]
```

The preceding code can be written again using a lambda function, which generates the same output:

```
list_yr=[]
ch=raw_input('Do you want to enter a year? ')
while 1:
    if ch in ('y','yes','Y'):
        yr=raw_input('Enter a year:')
        list_yr.append(int(yr))
        ch=raw_input('Do you want to enter another year? ')
    else:
        break
leap_yrs=filter(lambda n:n%4==0,list_yr)
print 'You have entered %d leap years, they are: \
%s'%(len(leap_yrs),leap_yrs)
```

The `map()` Function

There can be situations in which you need to perform the same operation on all the items of a sequence. The `map()` function helps you do this. The function `map()` takes a sequence, passes each item contained in the sequence through a function, and returns another sequence containing all the return values. The syntax of the `map()` function is similar to `filter()`:

```
map(function,sequence)
```

For example, if you want to add 3 to all the elements in a sequence, you can use the `map()` function in the following way:

```
>>>map((lambda a:a+3),[12,13,14,15,16])
```

The output of this statement will be:

```
[15,16,17,18,19]
```

Another illustration to calculate the cubes of all the items in a sequence is as follows:

```
>>>map((lambda a:a**3),[1,2,3,4,5])
[1,8,27,64,125]
```

Yet another example that rounds off all the values in a sequence containing float values is as follows:

```
>>>map(round,[13.4,15.6,17.8])
[13.0, 16.0, 18.0]
```

Figure 5.2 illustrates the working of a simple `map()` function.

As is evident from Figure 5.2, the `map()` function applies the specified function on all the members of the sequence and returns the resulting sequence. You learned how the `map()` function works with a single sequence containing N elements. In a more complex case, the `map()` function can take multiple sequences, each containing the same number of elements. In such a case, the `map()` function applies the given function on the corresponding elements of all sequences and returns the result in a tuple.

For example, the following call to `map()` adds the corresponding elements of the two sequences.

```
>>>map(lambda a,b:a+b,[4,7,3],[2,6,8])
[6, 13, 11]
```

The preceding example has two sequences, `[4,7,3]` and `[2,6,8]`, each of size 3. Therefore, the values of and N and M are 2 and 3, respectively. Here,

```
func=a+b
seq1=[4,7,3]
seq2=[2,6,8]
```

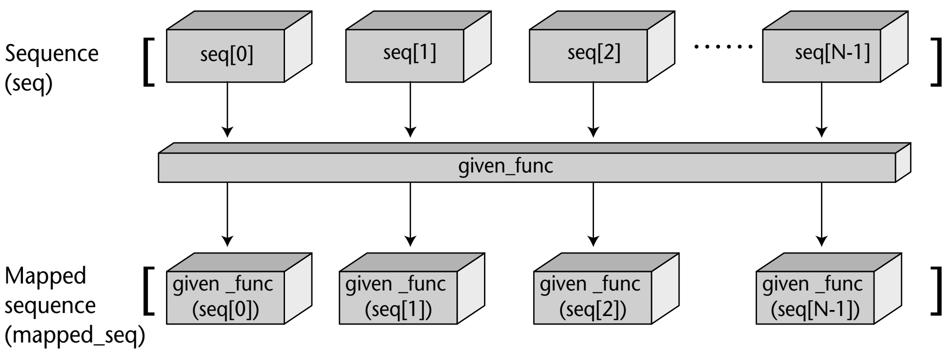


Figure 5.2 Working of the `map()` function.

then,

```
map(func,seq1,seq2)
=[4+2,7+6,3+8]
=[6,13,11]
```

Consider a few more examples where the `map()` function returns a tuple after applying the function on each element of the all sequences.

```
>>>map(lambda a,b:(a+b,a-b),[4,7,3],[2,6,8])
[(6, 2), (13, 1), (11, -5)]
```

Notice that in the preceding example, the mapped sequence contains tuples comprising the return values of the function. The first value in each tuple is the addition of corresponding values in the sequences, and the second value is their subtraction.

The `map()` function can also accept `None` as the function argument. In this case, `map()` will take the default identity function as its argument. The return value of the map function will contain tuples with one element from each sequence.

```
>>>map(None, [4,7,3],[2,6,8])
[(4, 2), (7, 6), (3, 8)]
```

Consider the following code that requests the user to specify integers and displays the square of all the values along with the original values.

```
def square(n):
    return n*n
seq=[]
ch=raw_input('Do you want to enter a number? ')
while 1:
    if ch in ('y','yes','Y'):
        inp=raw_input('Enter a number:')
        seq.append(int(inp))
        ch=raw_input('Do you want to enter another value? ')
    else:
        break
print map(None,seq,map(square,seq))
```

The sample output of the preceding code will be:

```
Do you want to enter a number? y
Enter a number:15
Do you want to enter another value? y
Enter a number:17
Do you want to enter another value? y
Enter a number:31
Do you want to enter another value? n
[(15, 225), (17, 289), (31, 961)]
```

In the preceding code, all the values entered by the user are stored in `seq`. The `map()` function is nested with another `map()` function. The inner `map()` function invokes the

square function and returns the sequence containing the squares of integers in seq. The outer map() function takes None as the function argument, which means that the resulting sequence will be the same as the sequence you passed in. Therefore, the result contains tuples with each value from the original sequence and the sequence containing squares.

You have learned about functions. You have also learned to pass parameters to functions. What are the rules that govern the use of variables inside and outside a specific function? To answer this question, let's learn about the scope of variables in the following section.

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable. The *scope of a variable* determines the portion of the program where you can access a particular identifier.

Global and Local Variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. This happens at the time when a local name is created for the object. This name survives until the function execution has completed. After this happens, that name is removed from the scope.

```
global_var=12
def ruf():
    local_var=34
    return global_var+local_var
```

In the preceding example, global_var is a global variable and local_var is a local variable. global_var is accessible inside both the main block of code and ruf() while local_str is accessible only inside ruf().

Number of Scopes

By its syntax, Python allows multiple levels of functional nesting. In other words, a variable declared inside a function is considered global to the function nested in the first function. Therefore, Python imposes multiple levels of scope. Consider the following example:

```
c='global'
def ruf():
    a='global for bee'
    def bee():
        b='local for bee'
        print a+b
```

```
        print c
    print a
    bee()
```

In the preceding code, `b` is local to `bee()` and `a` is local to `ruf()` but global to `bee()`. Therefore, `a`, `b`, and `c` can be accessed inside `bee()` whereas `a` and `c` can be accessed inside `ruf()`.

The lambda form also has the same scoping rules as the other functions. If the lambda form defines a new variable, then that variable is accessible only in the lambda form and not inside any other part of the program. This means that a function or a lambda form can access variables local to it, variables declared in levels above it, and global variables. Consider the following example:

```
c='global'
def ruf():
    a='a is global to lambda'
    bee=lambda b: a+b
    print bee('only lambda')
```

In the preceding code, `b` is the parameter passed to the lambda form. Therefore, `b` is local to the lambda form, but `a` and `c` are global for the lambda form.

Identify the Functions to Be Used

The following functions are used to generate login ids:

`isblank()`. This function ensures that the user does not miss entering the first name, the last name, and the date of birth.

`dobvalid_func()`. This function ensures that the user enters a valid date of birth, which includes a valid calendar month, day, and year.

`age_func()`. This function calculates the age based on the date of birth.

Write the Code

Let's write the code for the problem statement.

```
import time
def isblank(var):         #Function checks if the value passed in var is
blank
    while len(var)==0:  #and asks for another input
        print 'You can\'t leave it blank'
        var=raw_input("Enter a value: ")
    return var

def second(f):          #Takes object of first() as the parameter
    id=f+str(day)+str(month)    #and evaluates second value
    return id
```

```
def dobvalid_func():    #Checks if the date of birth is valid
    while 1:
        if year<=0 or month<=0 or day<=0:
            break
        if cur_year<year:   #Checks if current year
                           # is less than year of birth
            break
        if month>12:        #Checks if month of birth is greater than 12
           break
        if month in (1,3,5,7,8,10,12):#Checks if number of days in are
            if day>31:                #greater than 31 for applicable
                                      #month
                break
        elif month in (4,6,9,11):   #Checks if number of days in
                                    #date of birth
            if day>30:              #are greater than 31 for
                                    #applicable month
                break
        if year%4==0 and month==2:  #Checks if in a leap year,
                                    #number of days
            if day>29:              #in date of birth are greater than 29
                break               #for february
        return 1
    return 0


def age_func():                     #Calculates age based on date of
birth
    age=cur_year-year-1
    if month<cur_month or (month==cur_month and day<cur_day):
        age=age+1
    return str(age)
t=time.localtime(time.time())   #Determines the current time in a list
cur_year=t[0]                   #Extract the current year from the list
cur_month=t[1]
cur_day=t[2]
fname=raw_input("Enter your first name: ")
fname=isblank(fname)      #Call isblank function for fname

lname=raw_input("Enter your last name: ")
lname=isblank(lname)      #Call isblank function for lname
while 1:
    dob=raw_input("Enter your date of birth, mm-dd-yyyy: ")
    dob=isblank(dob)        #Call isblank function for dob
    if len(dob)<>10:
        print "Enter date in correct format!!"
        continue
    month=int(dob[:2])    #Extract month from date of birth
    day=int(dob[3:5])     #Extract day from date of birth
    year=int(dob[6:10])   #Extract year from date of birth
```

```
    if dobvalid_func()==0: #Checks if dobvalid_func returns true
        print "Invalid date of birth"
        continue
    else:
        break
print 'You can choose one of the following login names:'
first=fname+lname[0]    #Evaluates the first value
print "1. ",first
print "2. ",second(first)# Calls second() with first value as the
argument
third=lname[0]+fname+str(year)[2:]
print "3. ",third       #Evaluates the third value
fourth=fname[0]+lname+age_func()
print "4. ",fourth      #Evaluates the fourth value
```

Execute the Code

To be able to implement or view the output of the code to create login ids, you need to execute the following steps:

1. Write the preceding code in a text editor and save it with the `.py` extension.
2. At the shell prompt, type `python` followed by the name of the file if the file is in the current directory.
3. At the prompt `Enter your first name:`, enter `Laura`.
4. At the prompt `Enter your last name:`, press enter (leave it blank).
5. At the prompt `Enter a value:`, enter `Jones`.
6. At the prompt `Enter your date of birth, mm-dd-yyyy:`, enter `24`.
7. When you are prompted again to enter the date of birth, enter `12-24-1985`.

Figure 5.3 illustrates the output of the execution of the previous code.

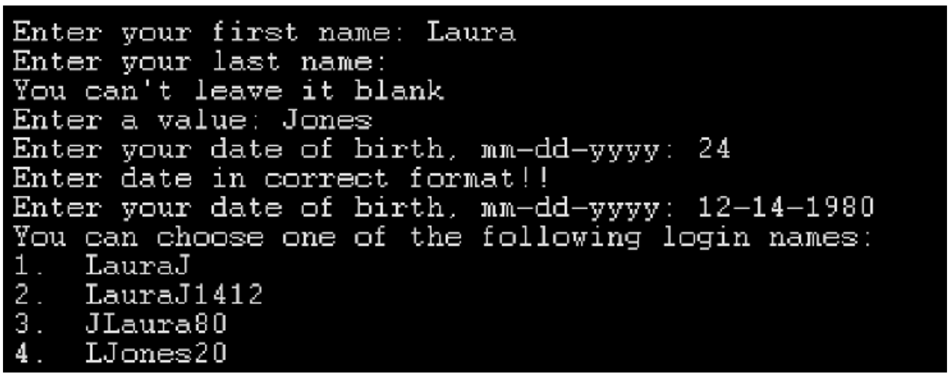


Figure 5.3 Output of the code.

Summary

In this chapter, you learned the following:

- A function is a block of organized reusable code that is used to perform a single, related action.
- Python supports the following types of functions:
  - User-defined functions
  - Built-in functions
  - Lambda forms
- The syntax for a function declaration is:

```
def functionname(parameters)
    function_docstring
    function_suite
```

- You can call a function by using the following types of formal arguments:
  - Required arguments
  - Keyword arguments
  - Default arguments
- *Required arguments* are the arguments passed to a function in correct positional order.
- Using *keyword arguments* in a function call identifies the arguments by parameter name.
- A *default argument* is an *argument* that takes a default value if a value is not provided in the function call for that argument.
- *Variable-length arguments* are not named in the function definition.
- You can also make a function return a value by using the `return` statement.
- Functions can be passed to other functions as arguments or can be elements of other objects.
- The `apply()` function basically invokes an object of another function when that function and its parameters are passed to the `apply()` function as arguments.
- The `filter()` function filters the items of a sequence based on a Boolean function and returns another sequence of the same type.
- The `map()` function performs the same operation on all the items of a sequence.
- The *scope of a variable* determines the portion of the program where you can access a particular identifier.

CHAPTER

6

Modules

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Use modules**
- **Import modules**
- **Use namespaces and scope variables**
- **Examine the module search path**
- **Test modules**
- **Use module built-in functions:**
  - `dir()`
  - `locals()`
  - `globals()`
  - `reload()`
- **Use packages**

Getting Started

In the previous chapter, you learned how functions provide a convenient way of designing programs that add modularity and overall programming simplicity to code. What happens when your code consists of many functions and it is typed in the Python interpreter itself? If you quit the interpreter, the function and variable definitions that you made are lost. In order to store your code for later use, you can write the code in a module.

In this chapter, you will learn how to organize code in Python modules. You will also learn how data is imported from modules into your programming environment. You will also understand how modules can be organized into packages.

Using Modules

Problem Statement

Techsity University's Web site consists of many pages containing forms that accept user details, such as details related to signing up, buying courses, and more. Every time a user enters details, he or she needs to be validated to determine whether the user has entered valid values. This is mandatory before carrying out further data processing. As a programmer, Jim is assigned the task of creating Python modules that perform these validations. These modules can be used to validate similar types of data whenever required. For example, many pages or forms on the University's Web site accept credit card numbers. The number needs to be validated before processing the user's requests. Jim will create a module that validates credit card numbers and can be imported whenever a user enters a credit card number.

Similarly, other modules can also be created for different types of user input. Initially, the University requires modules only to validate the first name, the last name, the date of birth, the quantity ordered by a user, and the credit card number.

Task List

- **Identify the modules to be used.**
- **Write the code for each module.**
- **Execute the code.**

Let's learn about modules to solve this problem of validating user input.

Modules

When writing code for complex tasks, it is quite common that the code becomes so large that you need to break it down into small pieces. In addition, there may be a function that you want to use in many programs without copying it into each program. You

can organize these small pieces of code into Python files, called *modules*, for easier maintenance. A module usually contains statements that have some relation with each other, such as a function and its related declarations or unrelated statements, which are written to perform a specific task. Modules can be made to interact with each other to access each other's attributes.

To be able to use a piece of code stored in a module, a module should be shared. The process of bringing in the attributes, such as variables and functions, of one module into another or at the interpreter itself is called *importing*. Importing is the basis of a module in Python. It is this feature that allows variables and functions declared in one module to be shared in different modules.

In other words, a module is a Python file containing Python definitions and statements. The filename is the name of the module with the `.py` extension appended. Here's an example of a simple module, `welcome.py`.

```
def print_func(a):
   print "Welcome",a
```

Use any text editor to write the preceding code and save it as `welcome.py`.

Importing Modules

You can import a module by using the `import` statement. The syntax of the `import` statement is this:

```
import module1[,module2[,... moduleN]
```

A module can be imported in another module or in the main module. The main module forms the top level of the collection of variables that you can access in a script and in the calculator mode.

When the interpreter encounters an `import` statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. We will discuss the search path later in this chapter.

To import the module `welcome.py`, type the following command in the interpreter:

```
>>>import welcome
>>>welcome.print_func('Jim')
```

The output of the preceding statement will be:

```
Welcome Jim
```

Here's another example of a module called `fib.py`, which generates the Fibonacci series.

```
#fib.py
def fibonacci(a,num1=0,num2=1):
    print num1
```

```
    while num2<a:
        print num2
        num2=num1+num2
        num1=num2-num1
```

In the Python interpreter, type the following command:

```
>>>import fib
```

This command does not execute the functions defined in the module directly; it only enters the module `fib` in the current symbol table. To access the function inside the module, use the following command:

```
>>>fib.fibonacci(100)
0
1
1
2
3
5
8
13
21
34
55
89
```

When a module is imported, the interpreter creates a byte-compiled version of the module. This byte-compiled version of the Python file has the `.pyc` extension, and this version is created in the same directory that contains the module. For example, for the module `fib.py`, the byte-compiled version will be `fib.pyc`.

Namespaces and Variable Scope

A namespace maps names with objects. When a name is added to a namespace by declaring a variable or a function, *binding* occurs and the name is said to have bound to the object. Similarly, the process of changing the mapping of the name with the object is called *rebinding*, and removing the mapping is called *unbinding*. At any given time during execution, there are only two or three active namespaces. These are local, global, and built-in namespaces. The names that can be accessed by the Python interpreter from these namespaces depend on the order in which the namespaces are brought into the system.

First, the built-in namespace, which consists of names in the `__builtins__` module, is loaded. Then, the global namespace for executing the module is loaded. When the module starts executing, the global namespace becomes the active namespace. If a function call is made during execution, the local namespace is created.

Therefore, namespaces involve the mapping of objects with their names. The scope of a name decides the locations within the code from which the name can be accessed.

For any attribute such as a variable or a function, the names within and outside the local namespace are in the local and global scope, respectively. In any program, local namespaces are created and deleted along with function calls, but the built-ins and global namespaces are permanent. When a module is imported for the first time, all the executable statements and function definitions are executed. Each module has its own private symbol table. This table is used as the global symbol table for all the functions inside that module. The global variables within a module can be used as any other global variables inside that module; however, these variables will be local to that module. For example, here is an example of a call to the function `fibonacci` that generates an error. The `fibonacci` function is local to the module `fib`.

```
>>>import fib
>>> fibonacci(100)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'fibonacci' is not defined
```

In order to access variables and functions local to a module outside that module, you need to use the name of the attribute following the module name. For example,

```
>>>import fib
>>>fib.fibonacci(100)
0
1
1
2
3
5
8
13
21
34
55
89
```

Reference to an object made using the dotted attribute notation is called *fully qualified name*. This notation prevents an exact conflicting match in the importing module's current namespace. The syntax for the notation is this:

```
modulename.functionname
```

For example, the function `fibonacci()` in the module `fib.py` is called `fib. fibonacci()`. There can be only one module with a given name that can be loaded on the Python interpreter. Therefore, `fib. fibonacci()` cannot conflict with another name. If there is another module named `fibnew.py` containing the function `fibonacci()`, the function will be called `fibnew. fibonacci()`. Therefore, there is no chance of conflict between the names of attributes.

If a function in a module returns a value instead of printing the value, the fully qualified name of a function can also be assigned to a variable. Consider the following example:

```
#casemod.py
def case(inp):
   if (inp >= 'A'):
      if(inp <= 'Z'):
         return 'Uppercase'
      elif (inp >= 'a'):
         if(inp <= 'z'):
            return 'Lowercase'
         else:
            return 'Input character > z'
      else:
         return 'Input character > z but less than a'
   else:
      return 'Input character less than A'
```

The preceding module checks whether an input variable is in uppercase or lowercase and returns a string. After importing the module, you can capture the value returned by the `case()` function in a variable as follows:

```
>>>import casemod
>>>c=casemod.case('H')
>>>c
'Uppercase'
```

The variable `c` has the value returned by the function `casemod.case()`. You can now use the variable `c` for executing this function.

More on Importing Modules

A module can be imported by another module. Python allows you to place an `import` statement, wherever required, in the importing module before using the attributes of the imported module. Conventionally, though, all `import` statements are placed at the beginning in the importing module. After importing a module, the attributes in the imported module are placed in the global symbol table of the importing module.

A variant of the `import` statement places the attributes of the imported module in the global symbol table of the importing module. Using the `from-import` statement, you can import specific elements from a module into your namespace. The syntax of the `from-import` statement is this:

```
from module import item1[,item2....[,itemN]]
```

For example, to import the function `fibonacci` from the module `fib`, use the following statement:

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module. In other words, when you import only names from other modules, the names become a part of the current namespace. When changes are made to these names in the importing module, only copies of the variables denoted by these names are altered. The original variables in the namespace of the imported module remain unaltered. Let's discuss this with the help of two modules, `importing_mod.py` and `imported_mod.py`.

```
#imported_mod.py
ruf='xyz'
def bee():
   print "ruf in importing_mod", ruf
#importing_mod.py
from imported_mod import ruf,bee
bee()
ruf='507'
print "ruf in imported_mod", ruf
bee()
```

When you run the script in `importing_mod.py`, you obtain the following output:

```
ruf in importing_mod xyz
ruf in imported_mod 507
ruf in importing_mod xyz
```

Notice that when the imported function name is called for the first time, the value of the variable `ruf` is the same as that assigned in the imported module originally. Even when the value of the variable `ruf` is changed in the importing module, it still remains the same in the imported module. This can lead to a conflict when you actually want to change the value of a variable in the importing module. The only solution to this is to use a fully qualified name by using the attribute dotted notation. Therefore, change the code of `importing_mod.py` as follows:

```
#importing_mod.py
from imported_mod import ruf,bee
imported_mod.bee()
imported_mod.ruf='507'
print "ruf in imported_mod", imported_mod.ruf
imported_mod.bee()
```

When you run the script in `importing_mod.py`, you obtain the following output:

```
ruf in importing_mod xyz
ruf in imported_mod 507
ruf in importing_mod 507
```

It is also possible to import all names from a module into the current namespace by using the following `import` statement:

```
from fib import *
```

NOTE "from module import *" provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly. After all the items are imported from a module by using `from module import *`, they become a part of the current namespace. This can lead to conflicts because the names from the imported module can clash with the names already present in the current namespace. The names from the imported module can even override the names that are already present in the current namespace.

If you do not want to import all the attributes in a module by using the "`from module import *`" statement, you can begin the name of the attribute with an underscore (_). In this way, you can hide data in your module even if you import all the attributes of the module. This technique is not useful, though, if the entire module is imported.

Python also contains a library of standard modules. Some modules are built into the interpreter. These provide access to the core of the language but are programmed to access operating system variables. Some of these modules are `sys`, `os`, and `time`. We will discuss some built-in modules as and when required in the later chapters of this book.

Module Search Path

While trying to import a module, you may have encountered the following error:

```
>>>import mymodule
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in ?
    import mymodule
ImportError: No module named mymodule
```

This happens because while importing a module, the Python interpreter searches for it in certain predefined locations. These predefined locations are a set of directories that constitute the Python *search path*. The search path contains the current directory and a set of other directories. If the interpreter cannot locate the module in the search path or if the search path is not set, it generates an error message. A default search path is automatically defined at the time of installation of the Python interpreter. The search path can also be modified to include other directories or remove directories already present in the search path.

The Python search path is specified in the environment variable PYTHONPATH. The syntax of PYTHONPATH is the same as that of the shell variable PATH. If PYTHONPATH is not set or if the interpreter does not find the module to be imported

in PYTHONPATH, the interpreter searches for the module in the installation-dependent default path, which is usually `.:/usr/local/lib/python` on Unix.

Actually, the module search path is stored in the system module sys as the `sys.path` variable. This variable contains a list of individual directory strings. The `sys.path` variable contains the current directory, PYTHONPATH, and the installation-dependent default. To view the directories in the `sys.path` variable, just import the sys module and type `sys.path` at the interpreter.

```
>>>import sys
>>>sys.path
```

The following output is obtained for a Linux computer. This output may vary depending on platforms and installation settings.

```
['','/usr/local/lib/python2.2/', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-tk', 'usr/local/lib/python2.2/
lib-dynload', '/usr/local/lib/python2.2/site-packages/']
```

If the module you need to import is stored in a directory that is not contained in the search path, you can modify the search path using the `sys.path` variable. The `sys.path` variable contains a list. Therefore, the list can be easily modified using standard list operations. For example,

```
>>>import sys
>>>sys.path.append('home/user/python/mod')
```

If you now see the contents of the `sys.path` variable, the variable will show the list containing the directory you specified.

```
>>>sys.path
['','/usr/local/lib/python2.2/', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-tk', 'usr/local/lib/python2.2/
lib-dynload', '/usr/local/lib/python2.2/site-packages',
'home/user/python/mod']
```

If there are many modules with the same name at multiple locations, then the Python interpreter will load the one it finds first while scanning through the search path sequentially.

Testing Modules

A module can be loaded in one of the following two ways:

- By directly executing it as a script
- By importing it in the main module or in other scripts

Loading a module executes code in the top-level portion of the module, which usually includes setting up of global variables and class and function declarations. In addition, any code inside the check for `__name__` built-in attribute of the module is

executed only when the module is executed directly as a script. We will discuss the `__name__` attribute a little later.

When a module is loaded by importing, it is loaded only once, even if it is imported multiple times. This means that variable and function declarations in the top level of the module happen only once, the first time a module is imported.

Modules, similar to all objects, have a built-in attribute, `__name__`. This attribute depends on how the module is being used. If you run the module directly as a script, then the `__name__` attribute will contain a special default value, `'__main__'`. If you import the module, then the `__name__` attribute will contain the name of the module without its path and file extension.

```
>>>import fib
>>>fib.__name__
'fib'
```

`__name__` can be used to design a test suite within the module itself by combining it in an if statement as follows:

```
if __name__='__main__':
true_suite
```

When you run the module directly as a script, the attribute `__name__` assumes the value `'main'`. Therefore, the suite inside the `if` statement is executed. If the `__name__` attribute is something else, the suite is ignored. This method of using a test suite with the `__name__` attribute is useful while testing and debugging modules before combining them in larger scripts.

Module Built-In Functions

Some built-in functions can be used in functional programming. We will describe some of them in this section.

dir()

The `dir()` built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables, and functions that are defined in a module. For example, you can import the built-in module `math` and the module `fib` defined earlier. You can also use the `dir()` function to determine the names defined by these modules. Let's see how this is done:

```
>>> import math, fib
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos',
'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp',
'log', 'log10', 'modf', 'pi', 'pow', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
>>>dir(fib)
['__builtins__', '__doc__', '__file__', '__name__', 'fibonacci']
```

If you do not supply any arguments to the `dir()` function, the `dir()` function lists the names you have defined currently.

```
>>> tup=('111','abc',64)
>>> str='xyz'
>>> import math, fib
>>> fibo=fib.fibonacci
>>> dir()
['__builtin__', '__builtins__', '__doc__', '__name__', 'fib', 'fibo',
'math', 'os', 'str', 'string', 'sys', 'tup']
```

The `dir()` function does not return the names of functions and variables that are loaded automatically by the interpreter. A list of functions and variables defined in the built-in standard module `__built-in__` can be displayed as follows:

```
>>>import ___builtin__
>>>dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'FloatingPointError', 'IOError', 'ImportError',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
'__import__', '__name__', 'abs', 'apply', 'buffer', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dictionary', 'dir', 'divmod', 'eval', 'execfile',
'exit', 'file', 'filter', 'float', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map',
'max', 'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'round', 'setattr',
'slice', 'staticmethod', 'str', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

`globals()` and `locals()`

The `globals()` and `locals()` functions can be used to return the names in the global and local namespaces depending on the location from where they are called. If `locals()` is called from within a function, it will return all the names that can be accessed locally from that function. If `globals()` is called from within a function, it will return all the names that can be accessed globally from that function. The return type of both these functions is dictionary. Therefore, names can be extracted using the `keys()` function. Let's discuss this with the help of two modules, `mainmod.py` and `fib1.py`.

```
# mainmod.py
import fib1
print "globals for main module:",globals().keys()
print "locals for main module:",locals().keys()
fib1.fibonacci(20)
# fib1.py
def fibonacci(a,num1=0,num2=1):
    print num1
    while num2<a:
        print num2
        num2=num1+num2
        num1=num2-num1
    print "globals for fibonacci:",globals().keys()
    print "locals for fibonacci:",locals().keys()
```

When you run the script in `mainmod.py`, you obtain the following output:

```
globals for main module: ['__builtins__', '__name__', 'fib1', '__doc__']
locals for main module: ['__builtins__', '__name__', 'fib1', '__doc__']
0
1
1
2
3
5
8
13
globals for fibonacci: ['__builtins__', '__name__', '__file__',
'__doc__', 'fibonacci']
locals for fibonacci: ['a', 'num1', 'num2']
```

Notice that both `locals()` and `globals()` return the same dictionary in the main module because all the names in the global namespace can be accessed locally as well. In other words, both global and local namespaces contain the same names in the main module.

reload()

You have learned that the code in the top-level portion of a module is executed only once when the module is imported. Therefore, if you want to reexecute the top-level code in a module, you can use the `reload()` function. The `reload()` function imports a previously imported module again. The syntax of the `reload()` function is this:

```
reload(module_name)
```

`module_name` is the name of the module you want to reload and not the string containing the module name. For example, if the module you want to reload is `mymodule`, then `module_name` should be named `mymodule` and not `'mymodule'`. In addition, `reload()` will load only the module that has been loaded fully and not by using the `from-import` statement.

In the previous chapter, you learned how functions add simplicity and modularity to your code. In this chapter, you learned how modules can be used to provide further modularity to your code. After writing several modules for a project, however, you may need to organize them in a structure so that you can easily access the names in each module.

Packages

In Python, packages allow you to create a hierarchical file directory structure of modules by using dotted module names. For example, `mymodule.mod1` stands for a module, `mod1`, in the package `mymodule`. You already know that the same function and variable names can be used in different modules. You use fully qualified names to address these functions and variables in the same module. Similarly, modules with the same names can exist in different packages and are referred with dotted module names. You can be certain that module names in multimodule packages will not conflict. The use of modules is a good practice to group related modules.

For example, a package can be created for handling a banking application. There are many different types of accounts, such as savings bank account, current account, and fixed deposit. You need to create modules that will create all these types of accounts. Then, you will create modules for different finances provided by the bank, such as corporate, personal, infrastructure, and project. In addition, you will write modules to perform services for the bank, such as advisory, custodial, ATM, and credit card. Like this, you can have an endless stream of modules to perform these operations. Here's a simple package structure that you can create for the previous example:

```
Bank/
   __init__.py
   Account/
      __init__.py
      savingsbank.py
      current.py
      fixed.py
   Finance/
      __init__.py
      personal.py
      corporate.py
      infrastructure.py
      project.py
   Services/
      __init__.py
      advisory.py
      custodial.py
      atm.py
      creditcard.py
```

The `__init__.py` file is required so that the interpreter can treat such a directory as a package. This file contains the information about the modules and subpackages that exist in a package so that subpackages are not hidden unintentionally while

searching for the module search path. In the simplest case, `__init__.py` can be empty but it can also contain initializer code for the module.

The `fixed.py` module in the previous example can be imported using the following:

```
import Bank.Account.fixed
```

To invoke a function in `fixed.py`, use the full name of the module:

```
Bank.Account.fixed.interest(Principal,rate=.12,period=5)
```

An alternative way of importing the submodule is:

```
from Bank.Account import fixed
```

The function interest can be invoked using the following statement:

```
fixed.interest()
```

You can also import the function or any variable in the `fixed` module directly by using the `from-import` statement as follows:

```
from Bank.Account.fixed import interest
```

Then, you execute it directly as follows:

```
interest()
```

Now let's identify the modules to be used for the problem statement in the beginning of the chapter to perform user validations.

Identify the Modules to Be Used

The following modules will be used to validate user input:

`isblank_mod.py`. This module ensures that the user does not leave any input for the required value blank.

`age_mod.py`. This module ensures that the user enters a valid date of birth, which includes a valid calendar month, day, and year. It also calculates the age based on the date of birth.

`qty_mod.py`. This module ensures that the quantity ordered is a numerical value and is a whole number.

`creditcard.py`. This module ensures that all the digits in the credit card number are numeric and that it is a 16-digit number.

`main_mod.py`. This module does not perform any validation but imports other modules and calls functions from them.

Write the Code

Let's now write the code.

The code for **isblank_mod.py** is as follows:

```
#isblank_mod.py
def isblank(var):
    #Function checks if the value passed in var is blank
    if len(var)==0:
        print 'You can\'t leave it blank'
    else:
        return 1
```

The code for **age_mod.py** is as follows:

```
import time
#Determines the current time in a list
t=time.localtime(time.time())
#Extract the current year from the list
cur_year=t[0]
cur_month=t[1]
cur_day=t[2]
#Checks if the date of birth is valid
def dobvalid_func(month,day,year):
    while 1:
        if year<=0 or month<=0 or day<=0:
            break
        #Checks if current year is less than year of birth
        if cur_year<year:
            break
        if cur_year==year and cur_month>month:
            break
        #Checks if month of birth is greater than 12
        if month>12:
            break
        #Checks if number of days are
        #greater than 31 for applicable month
        if month in (1,3,5,7,8,10,12):
            if day>31:
                break
        #Checks if number of days in date of birth
        #are greater than 30 for applicable month
        elif month in (4,6,9,11):
            if day>30:
                break
        #Checks if in a leap year, number of days
        #in date of birth are greater than 29
        #for february
        if year%4==0 and month==2:
            if day>29:
                break
```

```
        return 1
    return 0


def age_cal(dob):
    if len(dob)<>10:
        print "Date not in correct format!!"
        return 0
    m=int(dob[:2])    #Extract month from date of birth
    d=int(dob[3:5])   #Extract day from date of birth
    y=int(dob[6:10])  #Extract year from date of birth
    #Checks if dobvalid_func returns true
    if dobvalid_func(m,d,y)==0:
        print "Invalid date of birth"
        return 0
    else:
        age=cur_year-y-1
        if m<cur_month or (m==cur_month and d<cur_day):
            age=age+1
        return str(age)
```

The code for **qty_mod** is as follows:

```
# qty_mod.py
def qty_func(qtystr):
    #Checks if the quanity entered is numeric
     #and whole number
    i=0
    while i<len(qtystr):
        c=ord(qtystr[i])
        if (c in range(0,47) or c in range(58,255)) and c<>46:
            print "Please enter integers only"
            break
        else:
            i=i+1
    if i==len(qtystr):
        qty=float(qtystr)
        if qty<>int(qty) or qty<=0:
            print "Invalid value for quantity"
```

The code for **creditcard.py** is as follows:

```
# creditcard.py
def credit_func(cardno):
    #Checks if credit card no. is 16-digit and
    # contains only numbers
    if len(cardno)<>16:
        print "Credit card no. should be 16 digits"
    else:
        i=0
        while i<len(cardno):
```

```
            c=ord(cardno[i])
            if c in range(0,47) or c in range(58,255):
                print "Please enter integers only"
                break
            else:
                i=i+1
```

You need a module to verify that the validation modules that you have created execute properly. The code for this purpose in **main_mod.py** is as follows:

```
import isblank_mod
import qty_mod
import creditcard
import age_mod
fname=raw_input("Enter your first name: ")
#Call isblank function for fname
isblank_mod.isblank(fname)
lname=raw_input("Enter your last name: ")
#Call isblank function for lname
isblank_mod.isblank(lname)
date=raw_input("Enter your date of birth, mm-dd-yyyy: ")
#Call age_cal function from age_mod module
return_age=age_mod.age_cal(date)
if return_age<>0:
    print "Your age is %s years" %(return_age)
qty_order=raw_input("Enter quantity: ")
#Call isblank function for qty ordered
if isblank_mod.isblank(qty_order)==1:
    # Call function qty_func in qty_mod module
    qty_mod.qty_func(qty_order)
credit=raw_input("Enter credit card no.: ")
#Call isblank function for credit card no.
if isblank_mod.isblank(credit)==1:
    #Call function credit_func in creditcard module
    creditcard.credit_func(credit)
```

Execute the Code

In order to implement or view the output of the code and test validation modules, you need to execute the following steps:

1. Write the preceding code separately for each module in a text editor and save all the modules with the `.py` extension in the current directory.
2. At the shell prompt, execute `main_mod.py`.
3. At the prompt `Enter your first name:`, enter `Laura`.
4. At the prompt `Enter your last name:`, press enter (leave it blank).
5. At the prompt `Enter your date of birth, mm-dd-yyyy:`, enter `12-24-1984`.

6. At the prompt `Enter quantity:`, enter `6.3`.
7. At the prompt `Enter credit card no.:`, enter `12219461263`.

Summary

In this chapter, you learned the following:

- A *module* is a file containing Python definitions and statements used to organize code for easier maintenance. The filename is the name of the module with the `.py` extension appended.
- The process of bringing in attributes, such as variables and functions, of one module into another or into the main module is called *importing*.
- The syntax of the `import` statement is:

  ```
  import module1[,module2[,... moduleN]
  ```
- The global variables inside a module can be used as any other global variables; however, these variables will be local to that module.
- The reference to an object that is made using dotted attribute notation is called *fully qualified name*. This notation prevents an exact conflicting match in the current namespace importing module. The syntax for the notation is:

  ```
  modulename.functionname
  ```
- A module can be *imported* by another module. The syntax of the `from-import` statement is:

  ```
  from module import item1[,item2....[,itemN]]
  ```
- The `from-import` statement does not import the entire module. It imports only variables and functions from a module. Names imported in this way become a part of the current namespace. You alter a copy of the names, not the original.
- While importing a module, the predefined directories searched by the Python interpreter constitute the Python *search path*. The module search path is stored in the system module sys as the `sys.path` variable.
- Loading a *module* executes the code in the top-level portion of a module, which usually includes setting up global variables and class and function declarations. If the module is executed directly as a script, any code inside the check for `__name__` built-in attribute of the module is executed. When the module is loaded by importing it, it is loaded only once even if it is imported multiple times.
- The `dir()` built-in function returns a sorted list of strings containing the names defined by a module.
- The `globals()` and `locals()` functions can be used to return the names in the global and local namespaces depending on the location from which they are called.
- The `reload()` function reexecutes the top-level code in a module.
- A *package* is a hierarchical file directory structure used to organize related-Python modules.

CHAPTER

7

Files

OBJECTIVES

In this chapter, you will learn to do the following:

- **Use file objects**
- **Use standard input and output methods**
- **Use methods of file objects**
- **Use methods of the `os` module**
- **Use methods of the `os.path` module**

Getting Started

In the previous chapters, you learned how to accept data from users and display it. What happens to this data when you quit the interpreter? All the user input values will be lost. How can you store data for future use? To store data for future use, you can redirect user input into a file. You can also read the contents of a file.

In this chapter, you will learn how to write and append data to a file. You will also learn how to read the contents of a file.

Using File Objects

Problem Statement

Techsity University allows students from all over the United States to register for instructor-led training courses. The University wants to store course details such as the code, title, duration, and fee in a file. As a programmer, Jim is assigned the task of storing the course details in the file. Jim needs to create a script that will allow him to store the course details in a file.

Task List

- ✔ **Identify the functions and methods to be used.**
- ✔ **Write the code to store course details in a file.**
- ✔ **Execute the code.**
- ✔ **Verify that all information has been entered correctly.**

Before helping Jim to solve this problem, let's learn about the functions and methods that help us perform file-related operations.

Identify the Functions and Methods to Be Used

You require certain methods and functions to store data into files and read data from files. Python library provides basic functions and methods necessary to manipulate files by default. Importing special modules for this purpose is not required. These functions and methods are available for all file objects.

File Objects

As you know, Python uses objects for all types of data storage. You can use file objects to access files in order to read and write contents. File objects have built-in functions and methods that help you access all types of files.

In Python, the base of any file-related operation is the built-in function `open()`. The `open()` function returns a file object, which you can use to perform various file-related actions.

The `open()` Function

You can use the `open()` function to open any type of file. The syntax for the `open()` function is given here:

```
file object = open(file_name [, access_mode][,buffering])
```

The file_name argument is a string value that contains the name of the file that you want to access. The other two arguments, access_mode and buffering, are optional arguments. access_mode determines the mode in which the file has to be opened. buffering specifies the type of buffering to be performed while accessing the file. You will learn more about access modes and buffering in the sections that follow. Python returns a file object if it opens the specified file successfully. The following code displays an example for the use of the open() function.

```
>>>fileobj=open('/home/testfile','r')
```

The preceding code opens the file testfile in the /home directory in the read mode. Here, the buffering argument is omitted to allow system default buffering. Let's now discuss the various modes in which a file can be opened.

Access Modes. The default file access mode is read (r). The other common access modes are write (w) and append (a). In order to open a file in the read mode, the file should be created earlier. You can use the write and append modes with existing files or new files. When you open an existing file in the write mode, the existing content of the file will be deleted and new content will be written from the beginning of the file. When you use the append (a) mode for accessing an existing file, the new content will be written from the existing end-of-file position. If you access new files with the write and append modes, the files will be automatically created before writing data.

In addition to these modes, there are other modes for accessing files in binary mode. Windows and Macintosh operating systems treat text files and binary files differently. When you access files for reading or writing on Windows and Macintosh operating systems, the end-of-line characters will be changed slightly. Although the automatic change of the end-of-line characters in the file content does not affect ASCII files considerably, it will corrupt binary data in graphic and executable files.

NOTE Posix-compliant operating systems, such as Unix and Linux, treat all files as binary files. Therefore, there is no need to use binary mode explicitly for reading or writing in these operating systems.

In Windows and Macintosh systems, you can access files in binary mode by adding b to the normal access mode, such as rb, wb, and ab. For example, the following code will open testfile in binary mode for writing:

```
fileobj=open('c:/myfiles/testfile','wb')
```

You can access a file for both reading and writing by adding + with access mode, such as r+, w+, and a+. The following line of code will open testfile with read-write access.

```
fileobj=open('/home/testfile','r+')
```

Table 7.1 describes the use of different access modes.

Table 7.1 The Different File Access Modes

ACCESS MODE	DESCRIPTION
`r`	Opens a file for reading
`w`	Opens a file for writing
`a`	Opens a file for appending
`rb`	Opens a file for reading in binary format
`wb`	Opens a file for writing in binary format
`ab`	Opens a file for appending in binary format
`r+`	Opens a file for both reading and writing
`w+`	Opens a file for both writing and reading
`a+`	Opens a file for both appending and reading
`rb+`	Opens a file for both reading and writing in binary format
`wb+`	Opens a file for both writing and reading in binary format
`ab+`	Opens a file for both appending and reading in binary format

NOTE **If you omit the access mode argument in the `open()` function, the file will be opened in the read mode.**

Buffering. The `buffering` argument can take different integer values to specify the type of the buffering to be performed while accessing a file. If the buffering value is set to `0`, no buffering will take place. If the buffering value is `1`, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than `1`, then buffering action will be performed with the indicated buffer size. To specify default system buffering, you can either omit this argument or assign a negative value.

After the `open()` function returns a file object successfully, you can use the methods of file objects to perform different actions, such as writing and reading. Let's learn about the important methods of the file object.

Methods of File Objects

There are different types of methods, which you can use to read or write contents and move the cursor position within a file. First, let's learn about the methods, which are useful to write data to files.

Writing Data to a File

The `write()` method writes a string of data to a file. The string can be a set of characters in a single line or multiple lines or in a block of bytes. The `write()` method does

not insert line breaks automatically. If you want to insert line breaks, you add the NEWLINE character, `\n`, after each line. The following example displays the use of the `write()` method:

```
>>>fileobj=open('testfile','w')
>>>fileobj.write('This is the first line\n')
>>>fileobj.write('This is the second line\n')
```

In the preceding code, the first statement opens the file `testfile` for writing. Then, the second statement writes `This is the first line` in the file and adds a line break. Finally, the last statement adds another line with the text `This is the second line` and adds a line break. In both the second and third statements, line breaks are added due to the `\n` character.

TIP **You can insert a Tab character by using `\t`.**

The `writelines()` Method. You can use the `writelines()` method to write a list of strings to a file. The `writelines()` method also does not insert the NEWLINE character automatically. If you do not add the NEWLINE character at the end of each string in the list, the `writelines()` method will write the list items as a single string. The following example illustrates the use of `writelines()`.

```
>>> list=['one','two','three']
>>> i=0
>>> for x in list:
      list[i]=x+'\n'
      i=i+1
>>> fileobj=open('newtestfile','w')
>>>fileobj.writelines(list)
```

In the preceding code, the first statement creates a list consisting of three items. Then, the `for` loop adds the NEWLINE character `\n` to each item in the list. Next, the `open()` function opens the file `newtestfile` for writing, and the `writelines()` method writes the items of the list in `newtestfile`. The items of the list will be written as separate text line due to the NEWLINE characters.

Reading Data

You can read the data from a file by using the `read([size])` method. The `read()` method reads the bytes in a file to a string. The size argument is optional, which specifies the number of bytes to be read. If you do not specify the size, the value of this argument will be set to `-1` in order to read the entire contents of the file. The `read()` method also displays the NEWLINE characters. The following example displays the use of the `read()` method without specifying the size argument.

```
>>> fileobj=open('testfile','r')
>>>fileobj.read()
'This is the first line\nThis is the second line\n'
```

Now, let's look at the use of the size argument in the `read()` method.

```
>>> fileobj1=open('newtestfile','r')
>>>fileobj1.read(3)
'one'
```

The `read()` method reads the number of bytes from the current cursor position. For example, `newtestfile` has 14 bytes, and you can read the first 4 bytes by using the code `fileobj.read(4)`. When you use the `read()` method the next time, the method will start reading the contents from the fifth byte. Look at the following code:

```
>>> fileobj=open('newtestfile','r')
>>>fileobj.read(4)
'one\n'
>>>fileobj.read()
'two\nthree\n'
```

Here, the second code line reads the first 4 bytes. After that, when you execute the next read statement, the `read()` method returns all the bytes starting from the fifth byte in the file.

Two more methods help you read data from files, `readline()` and `readlines()`. Now, let's examine the functioning of these methods.

The `readline()` Method. You can use the `readline()` method to read a single line of text in a file. This method includes the NEWLINE character in the returned string. The code given here displays the functioning of `readline()`:

```
>>> fileobj2=open('testfile','r')
>>>fileobj2.readline()
'This is the first line\n'
```

The `readlines()` Method. You can also use the `readlines()` method to read the contents of a file. This method returns each text line in a file as a string in a list. Let's now use the `readlines()` method to read the contents of `testfile`:

```
>>> fileobj=open('testfile','r')
>>> fileobj.readlines()
['This is the first line\n', 'This is the second line\n']
```

The `readlines()` method also returns data from the current cursor position. Look at the following example:

```
>>> f=open('testfile','r')
>>> f.readline()
'This is the first line\n'
>>> f.readlines()
['This is the second line\n']
```

In this case, first the `readline()` method displays the first text line in `testfile`. After that, the `readlines()` method returns the remaining line, which is the second line.

Standard Input and Output

When you start Python, the system provides three standard files: `stdin`, `stdout`, and `stderr`. The file `stdin` contains the standard input, for which characters are normally entered using the keyboard. The file `stdout` has the standard output, which is the display on the monitor. The error messages generated by any code will be directed to the `stderr` file. The standard files are part of the `sys` module, and you need to import the `sys` module before accessing the standard files.

When you print a string, you actually write that string to the `stdout` file. When you receive data by using the `raw_input()` method, the `raw_input()` method reads the input from the `stdin` file. The standard files also support the methods for writing and reading data.

There are certain similarities and differences between the `print()` method and the `write()` method of the `stdout`. Let's look at the following examples to understand the functioning of `print()` method and `stdout.write()` methods.

- `print()` method

```
>>> print 'Welcome to Python'
Welcome to Python
```

- `stdout.write()` method

```
>>> import sys
>>> sys.stdout.write('Welcome to Python\n')
Welcome to Python
```

Both the examples display the text `Welcome to Python`. In the `stdout.write()` method, though, you have to add `\n` explicitly to indicate the end of the line.

Now, let's look at the functioning of the `raw_input()` method and the `stdin.readline()` method.

- `raw_input()`

```
>>> name=raw_input('Enter your name: ')
Enter your name:
```

- `standard_read.py`

```
import sys
sys.stdout.write('Enter a your name: ')
name=sys.stdin.readline()
sys.stdout.write(name)
```

Both the examples store the name entered by a user in the variable name and display the same. When you use the `stdin.readline()` method to accept a string to a variable, however, you have to write the code inside a file and then execute that file. If you write the code on the command prompt directly, you cannot store the value to a variable.

Supported Methods of File Objects

In addition to the methods of writing and reading data, file objects have certain other methods that help you perform different tasks on files such as moving within a file,

finding the current cursor position, and closing files. These methods include the following:

- `seek()`
- `tell()`
- `close()`

Let's discuss each of these methods in detail.

The `seek()` Method. You can use the `seek()` method to move the cursor position to different locations within a file. The syntax of the `seek()` method is this:

```
file_oobject.seek(offset,from_what)
```

The `seek()` method has two arguments, `offset` and `from_what`. The `offset` argument indicates the number of bytes to be moved. The `from_what` argument specifies the reference position from where the bytes are to be moved. Table 7.2 describes the values that can be taken by the `from_what` argument.

The `seek()` method is very useful when a file is opened for both read and write access. After writing data to a file, the current position of the cursor will be at the end of the file. In such a case, if you execute the `read()` method, Python returns a blank line. Here, you can use the `seek()` method to move the cursor to the beginning of the file and then read data. Consider the following example:

```
>>> fileobj=open('seekfile','w+')
>>> fileobj.write('Welcome to Python\n')
>>> fileobj.read()
''
>>> fileobj.seek(-18,1)
>>> fileobj.read()
'Welcome to Python\n'
```

In this example, when you execute the second line of code, Python writes 17 bytes of data, including the NEWLINE character, to `seekfile`. After this task, the current cursor position will be on the next byte, 18, which is blank. Therefore, the `read()` method returns a blank string. Then, to move the cursor position to the beginning of the file, you set the offset value of the seek method to `-18` from the current cursor position. When you execute the `seek()` method with these values, the cursor position moves to the byte zero. Now, the `read()` method displays the entire contents of the file from the beginning.

Table 7.2 The Values of the `from_what` Argument in the `seek()` Method

VALUE	DESCRIPTION
0	Uses the beginning of the file as the reference position
1	Uses the current position as the reference position
2	Uses the end of the file as the reference position

The `tell()` Method. The `tell()` method displays the current position of the cursor in a file. This method is helpful for determining the argument values of the `seek()` method. The following example illustrates the use of the `tell()` method:

```
>>> fileobj=open('tellfile','w+')
>>> fileobj.write('Welcome to Python\n')
>>> fileobj.tell()
18L
>>> fileobj.seek(-18,1)
>>> fileobj.tell()
0L
```

The `close()` Method. You can use the `close()` method to close access to a file. Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the `close()` method to close a file.

You have learned to access files for reading and writing data. In addition, various tasks are related to files, directories, and the file system. Now, let's discuss the file system and the various methods that help perform file- and directory-related tasks.

File System

Python has separate modules for different operating systems, such as `posix` for Unix, `nt` for Windows, and `mac` for Macintosh, to perform file- and directory-related tasks. The use of methods in these modules is slightly complex, though. The `os` module provides methods, which are simple to use, for performing file- and directory-related tasks. The `os` module acts as a front-end module to a pure operating system-dependent module. This module eliminates the direct use of operating system-dependent modules by loading appropriate modules according to the operating system installed on a computer.

You can divide the methods in the `os` module into three categories:

- File processing
- Directory
- Permissions

Let's discuss these methods in detail.

File-Processing Methods

The `os` module provides methods that help you perform file-processing operations, such as renaming and deleting files. You can rename files by using the `rename()` method and delete files by using the `remove()` method. Let's look at the functioning of these methods.

The `rename()` Method. The `rename()` method takes two arguments, the current filename and the new filename. The syntax for the `rename()` method is:

```
rename(current_file_name, new_file_name)
```

The following example renames `myfile` to `newfile`.

```
>>>import os
>>>os.rename('myfile', 'newfile')
```

When you execute this code, the `os` module converts this method to the appropriate rename command based on the operating system installed.

The `remove()` Method. You can use the `remove()` method to delete files by supplying the name of the file to be deleted as the argument. The following code displays the use of the `remove()` method.

```
>>>import os
>>>os.remove('newfile')
```

Directory Methods

The `os` module has several methods that help you create, remove, and change directories. You can also use directory methods to display the current directory and list the contents of a directory.

The `mkdir()` Method. You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created. The following code creates a directory called `newdir`:

```
>>> import os
>>>os.mkdir('newdir')
```

NOTE **The `os` module has one more method that allows you to create directories—`makedirs()`. This method also takes the name of the directory to be created as the argument.**

The `chdir()` Method. You can use the `chdir()` method to change the current directory. The `chdir()` method takes an argument, which is the name of the directory that you want to make the current directory. For example, you can change the current directory from home to the directory `newdir` by using the following code:

```
>>> import os
>>>os.chdir('newdir')
```

The `getcwd()` Method. The `getcwd()` method displays the current working directory. The following code displays the use of the `getcwd()` method:

```
>>> import os
>>> os.getcwd()
'/home/newdir'
>>>os.chdir('/home')
>>> os.getcwd()
'/home'
```

The `listdir()` Method. You can display the contents of a directory, which comprises files and subdirectories, by using the `listdir()` method. This method takes the name of the directory for which the contents are to be displayed as the

argument. For example, you can display the contents of the directory `newdir` by using the following code:

```
>>>import os
>>> os.listdir('newdir')
['File1', 'File2']
```

The last line in this code is the result obtained by executing the second line of code.

The `rmdir()` Method. The `rmdir()` method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should be removed. You can delete `newdir` by using the following code:

```
>>> import os
>>>os.chdir('newdir')
>>>os.remove('File1')
>>>os.remove('File2')
>>>os.chdir('/home')
>>>os.rmdir('newdir')
```

NOTE **You can also delete directories by using the `removedirs()` method of the `os` module. In this method also, you need to supply the name of the directory to be deleted as the argument.**

Permission Methods

The permission methods of the `os` module allow you to set and verify the permission modes. Table 7.3 describes the different permission methods.

The `os.path` Module

The `os.path` module includes functions that are useful to perform path-related operations. You can access the `os.path` module from the `os` module. The methods available in this module are useful for operations such as file path inquiries and retrieving information about files and directories. Let's look at the most useful methods of the `os.path` module.

Table 7.3 The Access Permission Methods of the `os` Module

METHOD	DESCRIPTION
`os.access(path,mode)`	This method verifies the access permission specified in the `mode` argument to the specified path. If the access permission is granted, the `access()` method returns `1`. Otherwise, this function returns `0`.
`os.chmod(path, mode)`	This method changes the access permission of the `path` to the specified mode.
`umask(mask)`	This method sets the mask specified as the argument and returns the old mask.

The `basename()` Method. The `os.path.basename()` method takes a path name as an argument and returns the leaf name of the specified path. For example, you have created a file called `file1` in a directory called `user1` under the `home` directory. Now, you can use the `basename()` method to retrieve only the filename from the path `/home/user1/file1` by using the following code:

```
>>> import os
>>>os.path.basename('/home/user1/file1')
'file1'
```

The `dirname()` Method. You can use the `os.path.dirname()` method to retrieve the directory name from a path name. For example, the following code returns the directory name of the path `'/home/user1/file1'`:

```
>>> import os
>>> os.path.dirname('/home/user1/file1')
'/home/user1'
```

The `join()` Method. The `os.path.join()` method joins two or more path components into a single path name. This method takes the path components as arguments. The following example illustrates the use of the `join()` method:

```
>>>import os
>>>current_dir=os.getcwd()
>>> print current_dir
/home/
>>>join_dir=os.path.join(current_dir,'testfile')
>>>print join_dir
/home/testfile
```

This example joins the path of the current working directory with the second argument, `testfile`, and returns the joined path.

The `split()` Method. The `os.path.split()` method splits a path, which is passed as an argument, into a directory name and a base name and returns them as a tuple. The following example splits the joined path obtained in the previous example:

```
>>> import os.path
>>> os.path.split(join_dir)
('/home','testfile')
```

The `splitdrive()` Method. The `splitdrive()` method is used to split the drive name and the path name of the argument passed in the method. The following examples illustrate the use of the `splitdrive()` method in both Unix and Windows versions:

- Unix version

  ```
  >>> import os.path
  >>> os.path.splitdrive('/home/testfile')
  ('', '/home/ testfile')
  ```

- Windows version

  ```
  >>> import os.path
  >>> os.path.splitdrive('c:/Python')
  ('c:', '/Python')
  ```

Table 7.4 The Information Methods of the `os.path` Module

METHOD	DESCRIPTION
getsize(file_name)	Returns the size of a file in bytes
getatime(file_name)	Returns the time when a file was last accessed
getmtime(file_name)	Returns the time when a file was last modified

The `splitext()` Method. The `splitext()` method separates the first name and the extension name of a filename. Consider the following code:

```
>>> import os.path
>>> os.path.splitext('testfile.txt')
('testfile', '.txt')
```

The Information Methods

The `os.path` module has three methods that allow you to retrieve information about the file size and file access. Table 7.4 describes these methods.

The following example displays the functioning of these methods:

```
>>> import os.path
>>> os.path.getsize('testfile')
47L
>>> os.path.getatime('testfile')
1006535165
>>> os.path.getmtime('testfile')
1006541232
```

Other Useful Methods

The `os.path` module also has certain methods that help to determine the existence of path names, directories, and files. Table 7.5 describes the important methods of the inquiry category.

Table 7.5 The Inquiry Category Methods of the `os.path` module

METHOD	DESCRIPTION
exists(path_name)	Returns `1` if `path_name` exists and `0` if `path_name` does not exist
isdir(path_name)	Returns `1` if `path_name` is a directory and `0` otherwise
isfile(path_name)	Returns `1` if `path_name` is a file and `0` otherwise

Now, let's discuss how we can use these methods.

```
>>> import os.path
>>> os.path.exists('/home')
1
>>> os.path.exists('/home/testfile')
1
>>> os.path.isdir('/home')
1
>>> os.path.isdir('/home/testfile')
0
>>> os.path.isfile('/home')
0
>>> os.path.isfile('/home/testfile')
1
```

In this example, the `exists()` method returns `1` in both the cases because both the paths exist. The `isdir()` methods return `1` and `0`, and the `isfile()` methods return `0` and `1` because `/home` is a directory and `testfile` is a file. You have learned about all the important file objects. Now, let's decide the methods to be used to solve the problem statement given in the beginning of the chapter.

Result

You need to store course details to a file and display the contents of that file. To do this, you need to use the following methods:

- `open()`
- `write()`
- `read()`

You can store the details of a course in a list because you need to write the details of a course in a single line.

Write the Code to Store Course Details to a File

The code for the problem statement in the beginning of the chapter is as follows:

```
heading=['Code','Title','Duration','Fee']
fileobj=open('course_details','a')
import os.path
ans='y'
while (ans=='y'):
     course_code=raw_input('Enter the course code: ')
     course_title=raw_input('Enter the course title: ')
     course_dur=raw_input('Enter the course duration: ')
     course_fee=raw_input('Enter the course fee: ')
```

```
        details=[course_code,course_title,course_dur,course_fee]
        i=0
        for x in details:
             details[i]=x+'\t'
             i=i+1
        j=0
        for x in heading:
             heading[j]=x+'\t'
             j=j+1
        if (os.path.getsize('course_details')==0):
             fileobj.writelines(heading)
             fileobj.writelines('\n')
        fileobj.writelines(details)
        fileobj.writelines('\n')
        ans=raw_input('Do you wish to add more records(y/n): ')
   fileobj.close()
```

Execute the Code

To view the output of the preceding code, the following steps have to be executed:

1. Type the code in a text editor.
2. Save the file as `read_data.py`.
3. Make the directory where you have saved the file the current directory.
4. On the Shell prompt, type:

```
python read_data.py
```

Verify the Solution

To verify that all information has been entered correctly, perform the following tasks:

1. Type the following code in a text editor:

```
fileobj=open('course_details','r')
output=fileobj.read()
print output
fileobj.close()
```

2. Save the file as `verify.py` in the same directory where you have saved the file `read_data.py`.
3. Make the directory where you have saved the file the current directory.
4. On the Shell prompt, type:

```
python verify.py
```

Summary

In this chapter, you learned the following:

- File objects are used to access files for reading and writing contents.
- The `open()` method is used to open any type of file.
- The access mode determines the mode in which a file has to be opened.
- The methods that are used to write data in a file are these:
  - `write()` Writes a string of data to a file.
  - `writelines()` Writes a list of strings to a file.
- The methods that are used to read the contents of a file are these:
  - `read()` Reads the bytes in a file to a string.
  - `readline()` Reads a single line of text in a file.
  - `readlines()` Returns the text lines in a file as strings in a list.
- There are three standard files in Python:
  - `stdin` This is the standard input file.
  - `stdout` This is the standard output file.
  - `stderr` The error messages generated by any code will be generated to this file.
- The standard files are part of the `sys` module.
- The `seek()` method is used to move the cursor position to different locations within a file.
- The `tell()` method displays the current cursor position in a file.
- The `close()` method is used to close access to a file.
- The `os` module acts as a front end to the operating system-specific module to perform the file- and directory-related operations.
- The methods in the `os` module can be categorized into three categories:
  - File-processing methods
  - Directory methods
  - Permission methods
- The `os.path` module provides several functions that are used to perform path-related operations.

CHAPTER

8

Object-Oriented Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- Use classes
- Use class objects
  - Attributes
  - Data attributes
  - Functional attributes
  - Instances
- Implement classes
  - Composition
  - Derivation
- Use inheritance
  - Multiple inheritance
- Override methods
- Use built-in functions
- Use wrapping
  - Delegation

Getting Started

In the previous chapters, you learned about the basics of programming in Python. You also learned how to use functions, modules, packages, and files to create programs in Python. In real-life applications, though, you cannot always create a well-structured program by using only these components. This is where the concept of object-oriented programming (OOP) comes in handy. Python provides support for OOP, but it is not necessarily required for creating programs in Python. Before you learn about OOP and its components, let's recap some features of OOP.

Introducing OOP

With advances in technology in different fields, the items used in day-to-day life are becoming more complex. Consider the example of a telephone. Earlier, telephones were heavy and wired, and they could be used only from a fixed location, but with advances in technology, phones have become small and mobile. You can be in touch with the world from anywhere. These items have become small and easy to use, but simultaneously they have become complex. You can appreciate the complexity of these gadgets only when you try to look at the details of their design and working.

The same trend is seen for software. Software applications are becoming more complex with time for various reasons. Generally, the inherent complexity of software depends on the task that it is programmed to perform. All applications need not be complex. For example, a software application developed by a user for personal use may be less complex than those that are used for accounting, air traffic control, and power supply applications.

Complexities are associated with software development and cannot be ignored. It is necessary to keep producing more advanced and useful applications. For example, mobile and satellite phones were developed in order to produce technologically advanced products. Otherwise, the world would still be using only wired phones. Care should be taken, though, to prevent the complexities from affecting the functioning of software and users' understanding of it. These complexities need to be simplified to make software easy to understand, manage, and use.

One of the ways in which these complexities can be simplified is by breaking the system into manageable components and arranging them in a systematic way. For example, a personal computer, which is a complex machine, can be broken down into different components, such as a CPU, a VDU, and a keyboard. The CPU can be further broken down into components, such as a processor, a clock chip, and memory. This demonstrates how a complex thing, such as a computer, can be broken down to its last logical component. A component can be defined and explained so that its functions and working can be understood. The functions of all the components, combined together, define the functionality of a computer. It is easy to understand the functions of a computer by knowing about the functions of its components and how they interact with each other. This approach gives a user a clear picture about a product or a concept.

Due to its benefits, this approach is now implemented in software development. This is known as the object-oriented approach to developing software. People have started developing software applications in the way you build high-rise buildings, by

putting together different components, big and small. It has led to the creation of complex software applications with much less effort and fewer lines of code.

Before we discuss OOP in Python, let's become familiar with the basic components of OOP.

Components of OOP

In earlier times, programs were written with the code arranged in a sequence of steps. Today, programming has become more structured and organized. Now, you can have code organized in logical blocks with specific functionality, which can also be reused. This logical method of programming gives you the freedom to create an object that meets your exact requirements and specifications.

Objects

Everything in this world is an object. Objects are of different shapes, sizes, and colors, and they have different purposes. You learn to recognize an object soon after birth. As you grow older, you start recognizing an object as an entity that has a definite boundary and a distinct shape, such as a book, a table, or a fan. After this, you start comparing objects to learn that each object is unique. Based on all these experiences, you can define an object as a tangible thing that displays some well-defined behavior. For example, a football is a tangible entity with a well-defined visible boundary. It has a unique purpose, and you can direct a specific action toward it, like kicking the football.

This concept of objects can be used in software development with a little refinement. An object is an entity with some physical characteristics, but for the purpose of software development, an object can also be anything that has a conceptual boundary. So, an object is an entity that has the following characteristics:

- It has a state.
- It might display a behavior.
- It has a unique identity.

Classes

The world is full of objects that have different characteristics and purposes. The task of managing these millions of objects is very difficult and needs planning. It requires a systematic approach to classification of objects, based on their characteristics. This is similar to the classification of living beings into kingdom, genus, family, and species. For example, the elephant, the bear, and the buffalo are all called animals. Why are they called so? All of them have some characteristics and the properties of the animal kingdom. For example, all of them have four limbs, give birth to their young, and have a solid bone structure. As the elephant, the bear, and the buffalo all share structural and behavioral similarities, they can be put in a class called animals.

Using the same fundamentals for the purpose of software development, you can identify different objects that have common attributes and define different classes. This

classification can be different for objects and classes in different situations. Let's consider the following scenario:

Dr. John Hanks is a doctor at the City Hospital. He wants to send a report about a critically ill patient for review to Dr. M. Smith, a renowned specialist at the Federal Hospital. Dr. Hanks is quite anxious that the report should reach Dr. Smith safely and on time.

As the report is very important and needs to be reviewed and returned as quickly as possible, Dr. Hanks has decided to send this report by hand through a trusted messenger. The doctor hands over the package to his messenger and instructs him to proceed with urgency.

Let's examine this situation and identify the objects therein. Identifying objects means finding objects that are relevant and central to the situation. In this scenario, one object involved is surely Dr. Hanks, who is sending something to someone through the other person. Apart from Dr. Hanks himself, the something (i.e., the report) and the someone (i.e., Dr. Smith) are also objects. There is also another object involved in this situation, the messenger. There are a total of four objects in the case: Dr. Hanks, the report, Dr. Smith, and the messenger.

Now, let's divide these objects into classes. Dr. Hanks, Dr. Smith, and the messenger can be part of the class, Living Things, and the report can be part of the class, Nonliving Things.

Later in the chapter, you will learn to use classes for programming in Python.

Benefits of OOP

There are many reasons for using the object-oriented approach to programming. One of the important reasons is that OOP maps to real-world situations. The object-oriented approach of programming is based on the object model, which provides the conceptual framework for OOP. As the world is full of objects, OOP models the real world correctly and provides a direct approach for solving real-world problems.

Another benefit of OOP is that you can reuse the classes that you create. This saves a lot of the time and energy spent on creating redundant lines of code. Let's examine an example.

Speedy Motors is a manufacturer of cars and sells them under the brand name Speedy. Speedy cars have not been doing well for some time. The marketing department has been very concerned about the plunging sales. The marketing staff plans to conduct a market study to find out why people do not like Speedy.

The results of the market study show that customers are happy with the engine and performance but do not like the body and colors of the car. After a detailed analysis of the results, the marketing team finds that the root cause of the problem is that the company's car doesn't have an identity of its own.

Based on the results of the study, the marketing department decides to give the car a new look and a new range of colors. The design department of the company creates a virtual design of the car and conducts a test using simulation software. The advertising department plans a promotional strategy based on the design created by the design department. At the same time, the finance department plans for the cost to design and manufacture the new car.

All these departments can be viewed as several teams working on one project. All the teams use the same information about the new car to arrive at a complete solution for the Speedy car.

The scenario at Speedy Motors can be used to explain the concept of reusability that is supported by the object-oriented approach. In the object-oriented approach, classes are designed such that they can be reused in several systems. In OO terms, the car is a class that can be used by all who need it.

Extending this scenario to include a situation in which Speedy Motors is planning to launch a new variant of the Speedy car, the work done on the design of the Speedy car can be used for the new variant. Not only can the attributes of the car, such as its length, width, and height, be reused, but also the process of designing can also be followed. For example, the process used by the finance department to compute the manufacturing cost of the Speedy car can be used for this new variant.

The software based on OOP is favorable to change. If, for some reason, you need to change few things in your software, you need not scrap your existing software and start developing a new one. You can simply update those parts that are affected by the proposed change and continue using the other parts of the software as they were earlier. This makes software based on object orientation easy to maintain and update. Let's look at the following case to explain this.

HighDesign is a company that creates computer graphics for the advertising business. All the internal systems of the company, such as accounts and payroll, are automated and use object-oriented techniques. The company has 500 employees; of them, 400 are graphic designers.

The company is now venturing into the business of software development. As the company is new in this business, it has now employed freelance software developers who work as temporary employees.

This new development in the company has rendered the old payroll system inadequate. The original payroll system was designed for two types of employees, confirmed employees and trainees. Though many attributes are common between confirmed employees and trainees, like name and address, some attributes are different. For example, a confirmed employee receives a base pay while a trainee receives a stipend.

The company now wants the system to be modified to accommodate freelancers as well. In the object-oriented system, this change does not mean that the entire payroll system needs to be revamped. A new class of freelancers needs to be introduced to take care of all the activities related to freelancers. The rest of the system remains unchanged.

Now that you are familiar with OOP and its components, let's look at a problem statement to learn to implement OOP using Python.

Using Classes

Problem Statement

Techsity University has already started with the process of automating its existing systems and making them available online. Working toward this goal, it is now planning to automate the library of the university and make it accessible online.

The library of Techsity University is very large and has various sections to cater to the needs of the reader. To start with the process of library automation, the university has decided to automate the book and software sections and make them available online. The success of this project will determine whether the rest of the library will be converted.

A team has been set up, with Sharon as the leader, to complete the task of library automation. The team has been given limited time to complete this project. In this short period, the team has to create a system that is fast, light, modular, and well structured. Apart from all this, the team has to take into consideration that two sections of the library are being computerized to start and that the rest of the sections will follow soon.

To make the code fast and light, they will have to keep the code short. They will also have to make parts of the code reusable, so that redundant lines of code are avoided. This would require them to write blocks of code that can be reused within the same application or with other applications, when the university plans to convert other sections of the library. For example, if a few fields, such as title and price, are common to books and software, the code to enter such information should be written only once in the application and should be accessible by other parts of the application when required.

The application should allow users to do the following:

- Add book and software records
- View book and software records
- Delete book and software records

To create an application that meets all the requirements, the project team has decided to take an object-oriented approach. They have come up with the following task list.

Task List

- ✓ **Identify the classes to be defined.**
- ✓ **Identify the class objects.**
- ✓ **Identify the classes to be inherited and their objects.**
- ✓ **Identify the methods to be overridden.**
- ✓ **Write the code.**
- ✓ **Execute the code.**

Identify the Classes to Be Defined

Classes are an important entity of object-oriented programming in Python. Python classes are a combination of the C++ and Module-3 classes. The Python class mechanism supports the most important features of classes. The terminology used in Python is different from the universally accepted terminology used for classes in C++ and

other languages. In Python, all the data types are objects and the word "object" need not mean an instance of a class, as in some other languages.

Python classes are data structures used to define objects. They contain data values and define behavioral characteristics. In Python, you define a class by using the keyword, class, followed by the class name in the header line. The suite of code follows the information in the header line. The syntax for a class declaration is this:

```
class Class_Name:
    'class_docstring'
    class_suite
```

The class suite contains statements that define the class. Classes can have multiple data types and functions. These class functions are commonly known as methods. They are defined as part of the class definition and are invoked by instances, which are required for executing classes. A class is generally defined earlier in the module so that its instances can be used in the code when required.

Classes have many benefits over the standard types, such as lists and directories. The major difference is that the standard types cannot be customized while classes can be customized and can have their own set of attributes. Another difference is that the definition of a class creates a new namespace, while standard data types do not provide a separate namespace. A namespace is a mapping from names to objects and is used as the local scope. All variable assignments happen in it. The function names are also bound to their definitions in this namespace. The standard data types have a common set of methods, and you can define methods as per your requirements in classes.

Result

In order to automate the books and software sections of the library, the following classes will be defined in the code:

`library.` This class will define the attributes and methods that the user will use to enter common information about books and software.

`books.` This class will define the attributes and methods that the user will use to enter specific information about books.

`software.` This class will define the attributes and methods that the user will use to enter specific information about software.

Identifying the Class Objects

As everything in Python is an object, classes are also objects. But, there is room for confusion here. When talking of classes as objects, it's important to understand that classes are not a realization of the objects that are defined in the class.

You can work with class objects by performing two types of operations. The first is creating attribute references, and the second is creating an instance of the class.

Class Attributes

A class attribute is an element of the class, which is referred to by using the standard dotted-attribute notation used for all attribute references in Python. The standard syntax for attribute references is this:

```
obj.name
```

In this code, `obj` refers to the name of the object and `name` refers to the name of the attribute.

The class attributes belong to the class in which they are defined. When you create a class, all the names that are in the namespace of the class at the time of its creation are considered as valid attribute names. These attributes can be either data attributes or functional attributes. Data attributes are created when they are assigned for the first time. A few Python data types use data attributes. For example, complex numbers use the `real` and `imag` attribute. The functional attributes are the methods, which are also used in other Python types, such as lists and dictionaries.

Data Attributes

Data attributes are commonly known as static members or class variables and are set at the time when the class is created. They can be used as any other variable. You can use methods to manipulate and update the variables in the class. These variables are directly linked to the class object and are not related instances. They are similar in nature to the `Static` variables used in C++. The most common attributes are the instance data attributes, as instances are the most commonly used objects in OOP. You generally define a class data attribute only when you want data types that are independent of instances and are static. Let's use an example to explain data attributes.

```
>>>class My_Class:
...     'An example of class data attributes'
...     a=0
...     b=1
...
>>>print My_Class.a
0
>>>My_Class.a=My_Class.a + My_Class.b
>>>print My_Class.a
1
```

In this example, `My_Class.a` and `My_Class.b` are the class data attribute references, returning an integer. `My_Class.a` is assigned another value by adding the values in `My_Class.a` and `My_Class.b`.

A class can have many attributes. It would be difficult to search for them in the code. Python provides a built-in function, `dir()`, which you can use to display a list of the names of the attributes currently contained in a class. You can also use the class dictionary attribute, `__dict__` to show the class attributes along with their values. `__dict__` is a special attribute and is available to all the classes. It consists of a dictionary of all the data attributes of a class. When you refer to a class attribute, the

__dict__ dictionary is searched for that attribute. If the attribute is found in the dictionary of the current class, it is returned. If the attribute is not found, then the dictionaries of the base classes of the current class are searched. Any changes made to the attributes of a class are reflected in the __dict__ attribute of only that class. dir() and __dict__ do not display all the built-in functions and variables of a class. Another way to show an output in the same way as the __dict__ attribute is to use the vars() built-in function. The function takes an instance of the class as an argument. You can also use vars() without an argument. If used in such a way, it returns a dictionary of the attributes and values corresponding to the current local symbol table.

Let's use the example of My_Class defined earlier. First, the dir() built-in function has been used to show the attributes of My_Class.

```
>>>dir(My_Class)
['__doc__', '__module__', 'a', 'b']
```

It lists all the attributes of My_Class. It also displays two special class attributes __doc__ and __module__. __doc__ contains the documentation string for the class, which is the first unassigned string that comes after the header line. It is similar to the document strings used for functions and modules. The documentation string for a class is specific to that class. __module__ contains the name of the module in which the class is defined. It is part of the fully qualified class name. The fully qualified name of the class, My_Class, is __main__.My_Class, in which __main__ is the module name. You can refer to these special class attributes as you refer to any other attribute.

Now let's use the __dict__ special class attribute to show the attributes of My_Class.

```
>>>My_Class.__dict__
{'a': 0, '__module__': '__main__', 'b': 1, '__doc__': 'An example of
class data attributes'}
```

It shows all the attributes of My_Class with their values, including the two special class attributes, __doc__ and __module__.

The second type of class attributes is functional attributes. The methods of a class are the functional attributes of that class. Therefore, functional attributes are also known as method class attributes. You need to create an instance of the class object before you can call a method. A method can be invoked only by using the instance of the class to which it belongs. Before you learn about functional attributes, let's learn how to create and work with class instances.

Class Instances

Other than attribute referencing, the operation that you can perform with classes is creating an instance of a class. A class instance is a variable that contains a reference to the class, which is of a data structure definition type. All instances are of the type, instance. In Python, it is easier to create an instance in comparison to other object-oriented languages. The process of creating an instance of a class is known as *instantiation*.

Class instantiation uses the function notation to call a class object, which creates an empty object. This object is then assigned to a local variable called the instance. Take the example of the class, `My_Class`, created earlier. Let's create a new instance of `My_Class` and assign the object to the variable, `m`.

```
>>>m = My_Class()
```

Now that you know about class instances, let's learn to work with functional attributes.

Functional Attributes

As discussed earlier, class attributes are of two types, data attributes and functional attributes. Functional attributes or method class attributes are the methods of a class. Working with functional attributes generally involves method handling. As methods are the functions defined in a class, they are defined as part of the class definition and are invoked by instances.

Python uses the concept of binding to restrict method calls by using only a class instance. Binding requires a method to be bound to an instance before you can call the method. Even if a method cannot be called directly by using the class object, it is still an attribute of the class in which it is defined. A method is considered bound if the instance is present and unbound if the instance is not present.

Let's consider an example to explain this concept. First, a new class, `My_Method_Class`, will be defined. This class will include a method, `my_method_example`. This method will then be called directly as any other function is called.

```
>>>class My_Method_Class:
... def my_method_example(self):
...     return 'An example of method reference'
...
>>>my_method_example()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in ?
    my_method_example()
NameError: name 'my_method_example' is not defined
```

When the `my_method_example` is called directly, the attempt fails and a `NameError` exception is raised. If the method is called by using the class object, it raises the `TypeError` exception. Let's try this:

```
>>>My_Method_Class.my_method_example()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in ?
    My_Method_Class.my_method_example()
TypeError: unbound method my_method_example() must be called with
My_Method_Class instance as first argument (got nothing instead)
```

In the previous examples, `my_method_example` is unbound. As it is a method and not a function available in the global namespace, it needs to be bound so that you can invoke it directly.

Let's now bind the my_method_example method and call it by using the instance of the class, My_Method_Class.

```
>>>m = My_Method_Class()
>>>m.my_method_example()
'An example of method reference'
```

In this example, the my_method_example method is bound by creating an instance of the My_Method_Class, m. This instance is then used to call the my_method_example method.

There are situations when you might want to invoke unbound methods. A few of the common reasons for doing so are when you want to apply static methods that are not supported by Python and when a specific instance of the method class is not available. You can call an unbound method by providing the object of the instance explicitly. The following code explains how you can do this.

```
>>>My_Method_Class.my_method_example(m)
'An example of method reference'
```

If the instance object is not available, you will not be able to call an unbound method.

The __init__() Constructor Method

The __init__() constructor is a special method available in Python that can be defined in a class to create objects in the initial state. If the class for which you are creating the instance contains any __init__() method, then it is invoked at the time of instantiation. The class object is returned, and the instantiation process is completed only after the __init__() method has been implemented. The __init__() method should not return any object other than the class object because this might lead to a conflict.

The __init__() method is also commonly used to set instance attributes. Instance attributes are data attributes, which contain values associated with a specific instance of a class. Instance attributes are not declared like local variables. They are set when the __init__() method is implemented at the time of instantiation. Instance attributes are referred to in the same way as any other data attribute.

Let's look at an example code to define the __init__() method in a class and initialize instance attributes:

```
def __init__(self):
    self.a=0
    self.b=0
```

The __init__() special method has self as the first argument like any other function and method defined in Python. During the instantiation operation, when the __init__() method is called, the object of the instance is passed to self. In the preceding example, two instance attributes, a and b, are also initialized. The __init__() method can also have arguments other than self. As the __init__() method is not invoked directly, you can pass values to these arguments through the instantiation operator at the time of instantiation.

Let's consider an example that will show how differently a class will behave after the `__init__()` method is defined in it. For this, a new class, `My_Init`, needs to be created on the basis of `My_Class` used earlier. Let's create the `My_Init` class:

```
>>>class My_Init:
... 'An example of __init__ method'
... def __init__(self, aVal, bVal):
...     self.a=aVal
...     self.b=bVal
...
>>>i = My_Init(1,2)
>>>i.a, i.b
(1, 2)
```

In this example, three arguments are defined in the `__init__()` method, `self`, `aVal`, and `bVal`. During the instantiation operation, the object of the instance is passed to `self`, and the values for the arguments, `aVal` and `bVal`, are passed as part of the class invocation call. The values in `aVal` and `bVal` are then assigned to the two instance attributes, which are accessed later. You can use the `dir()` built-in method and the `__dict__` special attribute to display all the instance attributes. Let's use the previous example and see how it works.

```
>>>dir(i)
['__doc__', '__init__', '__module__', 'a', 'b']
>>>
>>>i.__dict__
{'a': 1, 'b': 2}
```

Python also provides a special destructor method, `__del__`. Destructors are implemented in Python to do some processing after the references to all instance objects are removed and before the instances are deallocated. The `__del__` method is not implemented commonly because these conditions are difficult to meet.

Result

The `library` class will contain the following objects:

- Attributes

  `LibCode.` This attribute will contain the library code for the book or software.

  `Title.` This attribute will contain the title of the book or the software.

  `Price.` This attribute will contain the price of the book or software.

- Methods

  `init__().` This method is the constructor of the class. It will initialize the attributes defined in it.

  ```
  def __init__(self):
      'library class constructor'
  .....
  ```

```
.....
.....
```

lib_method(). This method will prompt the user to enter the item code, the title, and the price of a book or software.

```
def lib_method(self):
    'Enter common details for books and software'
.....
.....
.....
```

empty_file_method(). This method will delete the records of all the books or software.

```
def empty_file_method(self, FileName):
    'Delete book or software records'
.....
.....
.....
```

clear__screen_method(). This method will clear the screen.

```
def clear_screen_method(self):
    'Clear screen method'
.....
.....
.....
```

The `books` class will contain the following objects:

- Attributes

  Author. This attribute will contain the name of the author of the book.

  PageCount. This attribute will contain the total number of pages in the book.

  ISBN. This attribute will contain the International Standard Book Number (ISBN) of the book.

- Methods

  bks_method(). This method will ensure that the user enters all the details about a book.

```
def bks_method(self):
    'Enter book details'
.....
.....
.....
```

  bks_display(). This method will display the details of all the books.

```
def bks_display(self):
    'Display book details'
.....
.....
.....
```

The `software` class will contain the following objects:

- Attributes

  ProductOf. This attribute will contain the name of the software company.

  Size. This attribute will contain the size of the software.

- Methods

  sws_method(). This method will ensure that the user enters all the details about the software.

```
def sws_method(self):
    'Enter software details'
.....
.....
.....
```

  sws_display(). This method will display the details of all the software.

```
def sws_display(self):
    'Display software details'
.....
.....
.....
```

Identifying the Classes to Be Inherited and Their Objects

After identifying the classes and their attributes, you need to utilize these classes in your code. The following section discusses how you can utilize the classes that you have created.

Utilizing Classes

At the beginning of this chapter, you learned about the need for classes. Later, you learned how to create and work with them. Now, after you know all this, it is time to learn how to use classes in your program and make them a part of your code. It is important to fit classes in your program in a way that they follow the logical flow of your program. You can implement classes in your code in two ways, composition and derivation.

Composition

In composition, one class is made up of another. Classes are combined to create a code that provides better functionality. You can add classes inside other classes. This gives you the benefit of using the attributes and methods by using the original class objects. Composition also provides the benefit of code reusability.

Composition is useful if the classes have nothing in common and a class is just a required component of a larger class. If the relationship between two classes is close and they share common behavior, derivation is a better choice.

Derivation

Derivation provides a powerful feature of OOP, which allows for the use of the features and behavior of a class by another class without disturbing the rest of the program. It is possible for a dependent class to derive the features of its base class. These dependent classes are commonly called subclasses.

Subclasses and Inheritance

Inheritance is the property by which a subclass derives the attributes of the base class. The term “subclass” describes a class that inherits or derives the attributes from another class; the term “base class” describes a class from which a subclass has been derived. To understand this better, let’s relate this to the concept of parent and child. In inheritance, the base class is also termed as the *parent* and a subclass as the *child*. A child class can be a parent class for some other classes and so on. All classes higher than the parent class are termed as *ancestors*. This cycle of derivation can continue for multiple levels. This provides the benefit of code reusability.

The subclasses inherit most of the attributes of their base classes. Thus, a subclass has more attributes than its base class. A subclass can also modify some or all of the inherited attributes of the base class, but the base class cannot do anything with the attributes of the subclass. The syntax for declaring a subclass looks like this:

```
class Sub_Class_Name(Base_Class1[, Base_Class2, ...]):
    'class_docstring'
    class_suite
```

It is similar to the syntax used for declaring a base class, the only difference being that a list of all the base classes of the subclass is provided after the name of the subclass.

Let’s look at the following example to explain the concept of base classes and subclasses.

```
>>>class My_Base_Class:
... 'My_Base_Class is the parent class of My_Subclass'
... def my_base_class_method(self):
...     return 'Base class method'
...
>>>class My_Subclass(My_Base_Class):
... 'My_Subclass is the child class of My_Base_Class'
... def my_subclass_method(self):
...     return 'Subclass method'
...
>>>
>>>b = My_Base_Class()
>>>
>>>dir(b)
['__doc__', '__module__', 'my_base_class_method']
>>>
```

```
>>>
>>>s = My_Subclass()
>>>
>>>dir(s)
['__doc__', '__module__', 'my_base_class_method', 'my_subclass_method']
>>>
>>>s.my_base_class_method()
'Base class method'
>>>
>>>b.my_subclass_method()
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in ?
    b.my_subclass_method()
AttributeError: My_Base_Class instance has no attribute
'my_subclass_method'
```

In this example, two classes, `My_Base_Class` and `My_Subclass`, are defined. `My_Base_Class` is the parent class of `My_Subclass`. It defines one method, `my_base_class_method`. `My_Subclass` also defines one method, `my_subclass_method`.

The `dir()` built-in function is used to list the attributes of the class. The output of `My_Base_Class` shows `my_base_class_method`, but the output of `My_Subclass` shows both `my_base_class_method` and `my_subclass_method`. This shows that `My_Subclass`, which is the child class of `My_Base Class`, has inherited the attributes of the parent class.

As `My_Subclass` is a subclass of `My_Base_Class`, the `my_base_class_method` can be invoked by using `b`, which is an instance of the subclass, `My_Subclass`. But, when the `my_subclass_method` is invoked by using `s`, which is an instance of the base class, `My_Base_Class`, it results in an error. This shows that inheritance is only one-way; hence, only a subclass inherits the attributes of the base class and not vice versa.

Multiple Inheritance

When a class has one or multiple subclasses, it is known as inheritance. When a subclass is inherited from multiple base classes, it is known as *multiple inheritance*. All the subclasses derived from a single base class are termed as *siblings*.

When a class is derived from multiple classes, it is sometimes difficult to figure out the parent classes. Python provides you with a class attribute, `__bases__` that you can use to show the set of base classes for a subclass. `__bases__` is a tuple and displays only the parent class of a subclass and not all the ancestor classes.

Let's consider an example to explain multiple inheritance and the use of the `__bases__` attribute. This would require using the classes created earlier and defining two new classes, `My_Class_A` and `My_Class_B`. `My_Class_A` will be the subclass of the class, `My_Subclass`. `My_Class_B` will be the subclass of the classes, `My_Class_A` and `My_Base_Class`.

```
>>>class My_Class_A(My_Subclass):
... 'My_Class_A is the child class of My_Subclass'
... def my_subclass_method(self):
```

```
...     return 'Method of My_Class_A'
...
>>>My_Class_A.__bases__
(<class __main__.My_Subclass at 0x009462EC>,)
>>>
>>>class My_Class_B(My_Class_A, My_Base_Class):
... 'My_Class_B is the child class of My_Class_A and My_Base_Class'
... def my_subclass_method(self):
...     return 'Method of My_Class_B'
...
>>>My_Class_B.__bases__
(<class __main__.My_Class_A at 0x00955B84>, <class
__main__.My_Base_Class at 0x00936344>)
```

Even if `My_Class_A` is derived from `My_Subclass`, the result of `My_Class_B.__bases__` shows only the two parent classes, `My_Class_A` and `My_Base_Class`.

Result

The `library` class will be the base class, and both the `books` and `software` classes will be the subclasses of the `Library` class. The `library` class will define the attributes and methods that are common to both the books and software items, and it can be used by both the `books` and `software` classes. Both these classes will inherit all the objects of the `Library` class.

```
class library:
    'library class'
.....
.....
.....
class books(library):
    'books class'
.....
.....
.....
class software(library):
    'software class'
.....
.....
.....
```

Identify the Methods to Be Overridden

There might be times when you use the same names for the methods in the base class and the subclasses. This is known as method overriding. Method overriding is useful when you do not want to remember different method names. You might also use method overriding when you want to provide enhanced functionality in your subclasses. The following section elaborates on method overriding.

Method Overriding

In the earlier example, the classes `My_Subclass`, `My_Class_A`, and `My_Class_B` have a method with the same name, `my_subclass_method`. Let's use this example to explain method overriding. Let's first create instances of the classes, `My_Class_A` and `My_Class_B`.

```
>>>Aclass = My_Class_A()
>>>
>>>Bclass = My_Class_B()
>>>
```

Now, let's try invoking the `my_subclass_method` by using the instances of the classes, `My_Subclass`, `My_Class_A`, and `My_Class_B`.

```
>>>s.my_subclass_method()
'Subclass method'
>>>
>>>Aclass.my_subclass_method()
'Method of My_Class_A'
>>>
>>>Bclass.my_subclass_method()
'Method of My_Class_B'
```

In this example, `My_Class_A` defines `my_subclass_method()` and also inherits `my_subclass_method()` of `My_Subclass`. When you invoke `my_subclass_method()` by using the instance of `My_Class_A`, the inherited method of `My_Subclass` is not called. Instead, the `my_subclass_method()` of `My_Class_A` is called. In the same way, `My_Class_B` defines `my_subclass_method()` and also inherits `my_subclass_method()` of `My_Class_A`. But, when you invoke `my_subclass_method()` by using the instance of `My_Class_B`, the `my_subclass_method()` method of `My_Class_B` is called, not the method of `My_Class_A`.

The methods in the base classes override the methods of their subclasses when you want to apply static methods that are not supported by Python. When a specific instance, but when the methods of the subclass, is invoked by the instances of the subclass, the overriding methods (methods of the base class) are not invoked. In such a situation, if you want to invoke the methods of the base class, you can do that in the following way:

```
>>>My_Subclass.my_subclass_method(Aclass)
'Subclass method'
```

You call the method of the base class by invoking an unbound base class method and providing the instance of the subclass. The instance of `My_Subclass` is not required because the instance of `My_Class_A`, which is a subclass of `My_Subclass`, is available.

If the base class has a constructor `__init__()` and the subclass also has a constructor `__init__()`, the constructor of the base class is not inherited by the subclass. When you instantiate the subclass, the `__init__()` method of the subclass is automatically

invoked. You can invoke the `__init__()` method of the base class in the same way as you invoke the overridden method of a base class, by calling the unbound base class method and explicitly providing the instance of the subclass. Let's consider an example.

```
>>>class My_Class:
... 'My Class is the parent class of My_Child_Class'
... def __init__(self):
...     print 'My_Class constructor'
...
>>>class My_Child_Class(My_Class):
... 'My_Child_Class is the subclass of My_Class'
... def __init__(self):
...     print 'My_Child_Class constructor'
...
>>>
>>>MyC=My_Class()
My_Class constructor
>>>
>>>MyChC=My_Child_Class()
My_Child_Class constructor
>>>
```

In this example, when the subclass is instantiated, the constructor of the subclass is called. Now, let's add an explicit call to the constructor of the base class in the constructor of the subclass.

```
>>>class My_Child_Class(My_Class):
... 'My_Child_Class is the subclass of My_Class'
... def __init__(self):
...     My_Class.__init__(self)
...     print 'My_Child_Class constructor'
...
>>>
>>>MyChC=My_Child_Class()
My_Class constructor
My_Child_Class constructor
```

In this example, the instance of the subclass is explicitly passed when calling the constructor of the base class, `My_Class.__init__(self)`.

Result

The method names used by all the classes are unique, but they all have the constructor methods, `__init__()`, with the same name. The constructor method of the subclass will override the constructor method of the base class.

```
class library:
    'library class'
    def __init__(self):
        'library class constructor'
```

```
        .....
        .....
        .....
.....
.....
.....
class books(library):
    'books class'
    def __init__(self):
        'books class constructor'
        .....
        .....
        .....
.....
.....
.....
class software(library):
    'software class'
    def __init__(self):
        'software class constructor'
        .....
        .....
        .....
.....
.....
.....
```

Figure 8.1 explains the final class structure.

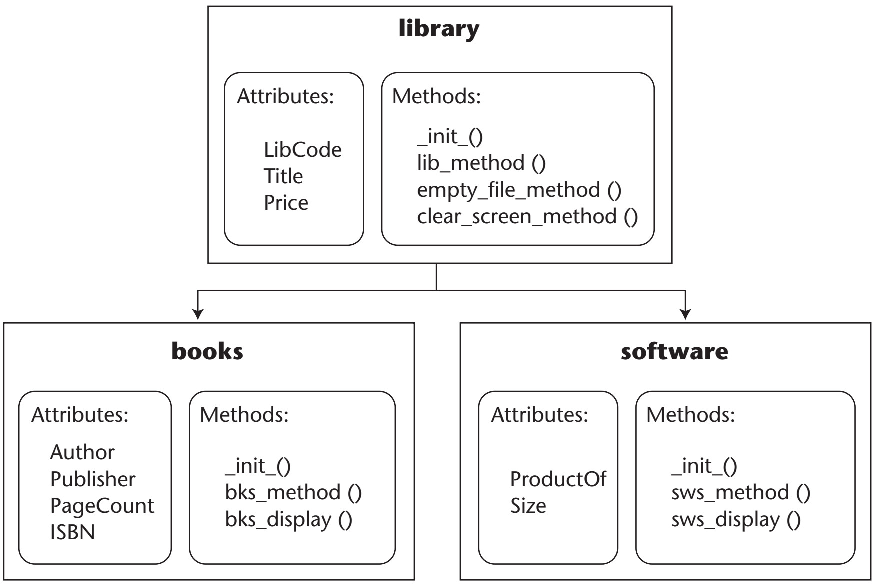


Figure 8.1 Class structure.

Using Built-In Functions

You have already discussed and used some built-in functions, such as `dir()` and `var()`. Let's now look at the syntax of some other common built-in functions available in Python for OOP and learn how to use them.

`isinstance(object1, object2)`

The `isinstance()` function takes two arguments, in which the first argument is an instance object and the second argument is a class object or a type object, for example, `object1` and `object2`.

Let's consider an example to explain the `isinstance()` function. Continuing with the same example that was used to explain inheritance, `My_Base_Class` already has an instance, `b`, and `My_Subclass` has an instance, `s`. Let's create instances for `My_Class_A` and `My_Class_B`.

```
>>>classA = My_Class_A
>>>
>>>classB = My_Class_B
```

In the `isinstance()` function, if the second argument is a class, the function determines whether `object1` is an instance of the class `object2`. If this is true, the function returns 1.

```
>>>isinstance(classA, My_Class_A)
1
```

If `object1` is not an instance of the class `object2`, the function returns 0.

```
>>>isinstance(classA, My_Class_B)
0
```

In the `isinstance()` function if the second argument is a type object, the function determines whether `object1` is of the type `object2`. If this is true the function returns 1. You use the `type()` function to determine the type of any object. In Python, the standard types need not be classes and they cannot be used for direct derivation. All the built-in standard types of Python are defined in the `types` standard module.

```
>>>isinstance('a', type('z'))
1
>>>
>>>type('a')
<type 'str'>
>>>
>>>type('z')
<type 'str'>
```

Here, both `'a'` and `'z'` are strings and are of the type `str`. Hence, the `isinstance()` function returns 1.

If `object1` is not of the type `object2`, the `isinstance()` function returns 0.

```
>>>isinstance('a', type(1))
0
>>>
>>>type(1)
<type 'int'>
```

Here, `1` is an integer and is of the type `int`, while `'a'` is a string and is of the type `str`. Hence, the `isinstance()` function returns 0.

If `object2` is not a class object or a type object, the function raises a `TypeError` exception.

```
>>>isinstance('a', classA)
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in ?
    isinstance('a', classA)
TypeError: isinstance() arg 2 must be a class or type
```

Here, `classA` is not a class object or a type object. Hence, the `isinstance()` function raises a `TypeError` exception.

issubclass(class1, class2)

`issubclass()` takes two arguments, and both of them are classes—for example, `class1` and `class2`.

It determines whether `class1` is a subclass of `class2`, and if this is true, the function returns 1. Let's take the same example that was used to explain inheritance.

```
>>>issubclass(My_Class_B, My_Base_Class)
1
```

Here, `My_Base_Class` is the parent class of `My_Class_B`.

If `class1` is not a subclass of `class2`, the function returns 0.

```
>>>issubclass(My_Subclass, My_Class_B)
0
```

Here, `My_Class_B` is not the parent or ancestor class of `My_Subclass`.

If `class2` is an ancestor of `class1`, the result will be true and the function will return 1.

```
>>>issubclass(My_Class_B, My_Subclass)
1
```

`My_Class_B` is inherited from `My_Class_A`, and `My_Class_A` is inherited from `My_Subclass`, thus making `My_Subclass` the ancestor class of `My_Class_B`.

Also, if both `class1` and `class2` are the same, the function returns 1 because a class is considered a subclass of itself.

```
>>>issubclass(My_Class_B, My_Class_B)
1
```

But, if one or both the arguments of the issubclass() function are not class objects, a TypeError exception is raised.

```
>>>issubclass(classB, My_Class_B)
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in ?
    issubclass(classB, My_Class_B)
TypeError: issubclass() arg 1 must be a class
```

Here, classB is not a class object but an instance object. Hence, the function raises a TypeError exception.

hasattr(obj, attr)

The hasattr() function takes two arguments, in which the first argument is an object (for example, obj) and the second argument is a string (for example, attr). It determines whether the string is the name of one or more of the object attributes. This function can be used to check whether the object attributes that you want to refer to actually exist.

Let's consider an example to explain the hasattr() function. Let's create a new class, My_Attr_Class, and define two class attributes.

```
>>>class My_Attr_Class:
    a=0
    b=1
```

If attr is the name of an attribute of the object, obj, the function returns 1.

```
>>>hasattr(My_Attr_Class, 'a')
1
```

Here, a is an attribute of the class, My_Attr_Class. Hence, the function returns 1. If attr is not the name of an attribute of the object, obj, the function returns 0.

```
>>>hasattr(My_Attr_Class, 'z')
0
```

Here, z is not an attribute of the class, My_Attr_Class. My_Attr_Class has only two attributes, a and b. Hence, the function returns 0.

getattr(obj, attr)

The getattr() function takes two arguments, in which the first argument is an object (for example, obj) and the second argument is a string (for example, attr). It returns the value of the attribute that has the same name as the string.
If attr is the name of an attribute of the object, obj, the function returns the value of the attribute.

```
>>>hasattr(My_Attr_Class, 'a')
0
```

Here, a is an attribute of the class, My_Attr_Class, and its value is 0.

If attr is not the name of an attribute of the object, obj, the function raises an AttributeError exception.

```
>>> getattr(My_Attr_Class, 'z')
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in ?
    getattr(My_Attr_Class, 'z')
AttributeError: class My_Attr_Class has no attribute 'z'
```

Here, z is not an attribute of the class, My_Attr_Class. My_Attr_Class has only two attributes, a and b. Hence, the function raises an AttributeError exception.

setattr(obj, attr, val)

The setattr() function takes three arguments, in which the first argument is an object (for example, obj), the second argument is a string (for example, attr), and the third argument is a value (for example, val). You use this function to change the value of an existing attribute or set a new attribute for an object.

When you use the setattr() function to change the value of an existing attribute, it assigns the value, val, to the attribute, attr, of the object, obj, or it sets a new attribute, attr, in the object, obj, and assigns the value, val.

Let's update the value of the attribute, a, of the class, My_Attr_Class, from 1 to 10.

```
>>>setattr(My_Attr_Class, 'b', 10)
>>>
>>>getattr(My_Attr_Class, 'b')
10
```

When you use the setattr() function to set a new attribute, it sets a new attribute, attr, in the object, obj, and assigns the value, val. Let's consider an example. Let's first list the present attributes of the class, My_Attr_Class, before the new attribute is added.

```
>>>dir(My_Attr_Class)
['__doc__', '__module__', 'a', 'b']
```

My_Attr_Class has only two attributes, a and b, besides the standard built-in attributes.

Let's now set a new attribute, c, with the value as 100 in the class, My_Attr_Class.

```
>>>setattr(My_Attr_Class, 'c', 100)
>>>
>>>getattr(My_Attr_Class, 'c')
100
```

Let's now list the attributes of My_Attr_Class to confirm that the new variable, c, has been set with the value, 100.

```
>>> My_Attr_Class.__dict__
{'a': 0, 'c': 100, '__module__': '__main__', 'b': 10, '__doc__': None}
c appears in the list with the value as 100.
```

delattr(obj, attr)

The `delattr()` function takes two arguments, in which the first argument is an object (for example, `obj`) and the second argument is a string (for example, `attr`). You use this function to delete an existing attribute from an object. It deletes an existing attribute, `attr`, from an object, `obj`.

Let's delete the attribute, `c`, of the class, `My_Attr_Class`.

```
>>>delattr(My_Attr_Class, 'c')
>>>
>>>dir(My_Attr_Class)
['__doc__', '__module__', 'a', 'b']
```

`c` is deleted from the class, `My_Attr_Class`.

delattr(obj, attr)

`delattr()` takes two arguments in which the first argument is an object (for example, `obj`) and the second argument is a string (for example, `attr`). You use this function to delete an existing attribute from an object. It deletes an existing attribute, `attr`, from an object, `obj`.

Let's delete the attribute, `c`, of the class, `My_Attr_Class`.

```
>>>delattr(My_Attr_Class, 'c')
>>>
>>>dir(My_Attr_Class)
['__doc__', '__module__', 'a', 'b']
```

`c` is deleted from the class, `My_Attr_Class`.

Wrapping

Python allows you to modify, add, or remove some functionality to an existing object, such as a data type or some code by packaging the object. This is known as wrapping.

Wrapping is important when you want to derive the behavior of a standard type. You need to wrap a type to derive its behavior because Python does not support the derivation of standard types. To derive a type, you wrap it as a member of a class and then use the object of this class. You can use the wrapped type to provide the behavior of the standard type as you desire, remove what you do not want, and also provide some improved functionality. Wrapping generally consists of customizing the existing type to provide some enhanced functionality over the existing behavior of a standard type. The wrapping of a class is also possible, but you can also wrap an object in the way you wrap a type.

Delegation

Delegation is a characteristic of wrapping that uses the existing functionality of the type to enable code reusability. Delegation takes advantage of the existing functionality of the type. In delegation, the existing functionality is delegated to the default attributes of the object, and a new class manages the extra functionality.

You implement delegation by overriding the `__getattr__()` method that contains a call to the `getattr()` function. When an object attribute is referred to, it is searched locally first in the local namespace and then in the class namespace. If it is not found at both the locations, the search for the original object begins by invoking the `__getattr__()` method, which in turn calls the `getattr()` function.

Write the Code

Let's write the code for the problem statement.

```
import os       #Imports the OS module, which would be required
                #to execute the system commands

ClearScreen = os.system('clear')    #Clears the screen as soon
                                    #as the code is executed

class library:      #Defines the library class, which is the
                    #top most class in the hierarchy
    'library class'
    def __init__(self):
        'library class constructor'
        LibCode=Title=Price=''      #Initializes the attributes
                                    #of the library class
        FileName=''

    def lib_method(self):       #Takes input for three
                                #attributes, LibCode, Title,
                                #and Price, which are common to
                                #books and software

        'Enter common details for books and software'
        LibCode=raw_input('Enter the library code: ')
        Title=raw_input('Enter the title: ')
        Price=raw_input('Enter the price (in $): ')
        return LibCode,Title,Price
    def empty_file_method(self, FileName):      #Accepts the
                                                #name of the
                                                #file and
                                                #empties it
        'Delete all book or software records'
        File=open(FileName,'a')
        File.seek(0,2)      #Goes to the end of file

        FileLen=File.tell()     #Stores the length of file in
```

```
                          #an attribute
        if FileLen == 0L:       #Checks if the length of file
                          #is zero
            print
            print
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print 'xxxxxxx FILE ALREADY EMPTY xxxxxxx'
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print
            print
        else:
            File.truncate(0)        #Empties the file, if the
                                    #lenght of file is not zero

            if FileName == 'BookDetails':       #Checks if the
                                                #filename is
                                                #BookDetails,
                                                #which contains
                                                #book records
                print
                print
                print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
                print 'xxxxxxx ALL BOOK RECORDS DELETED xxxxxx'
                print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
                print
                print
            else:
                print
                print
                print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
                print 'xxxxxxx ALL SOFTWARE RECORDS DELETED xxxxxx'
                print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
                print
                print
        File.close()
    def clear_screen_method(self):      #Clears the screen when
                                        #called within the code
        'Clear screen method'
        KeyInput=0
        while not KeyInput:
            print
            ch=raw_input('Press Enter to continue ')
            if ch!='':      #Checks if the input is not the
                            #Enter key
                print
                print
                print 'Wrong key pressed. You can only press Enter '
            else:
                ClearScreen = os.system('clear')
                KeyInput=1
class books(library):       #Defines the books class, which is
```

```
                        #a subclass of the library class
    'books class'
    def __init__(self):
        Author=Publisher=PageCount=ISBN=''      #Initializes the
                                                #attributes of
                                                #the books
                                                #class
    def bks_method(self):       #Takes input for book details
        'Enter book details'
        BkFile=open('BookDetails', 'a')     #Creates and opens
                                            #a file in the
                                            #append mode

        libM=self.lib_method()      #Calls the method of the
                                    #base class, which takes
                                    #input for three attributes,
                                    #LibCode, Title, and Price,
                                    #and returns their values

        BkFile.write(libM[0] + ',')     #Values in attributes
                                        #are written to the
                                        #file
        BkFile.write(libM[1] + ',')
        Author=raw_input('Enter the name of the author: ')
        BkFile.write(Author + ',')
        Publisher=raw_input('Enter the name of the publisher: ')
        BkFile.write(Publisher + ',')
        ISBN=raw_input('Enter the ISBN: ')
        BkFile.write(ISBN + ',')
        PageCount=raw_input('Enter the page count: ')
        BkFile.write(PageCount + ',')
        BkFile.write(libM[2] + '\n')
        BkFile.close()
        print '''
You have entered the following details for a book:
==================================================
Library code: %s
Title: %s
Author: %s
Publisher: %s
ISBN: %s
Page count: %s
Price: $%s''' % (libM[0], libM[1], Author, Publisher, ISBN, PageCount,
libM[2])     #Prints the book details entered
                      #recently

    def bks_display(self):      #Display all the book records
                                #available in the BookDetails
                                #file
        'Display book details'
        BkFile=open('BookDetails', 'a')
        BkFile.seek(0,2)
```

```
        BkFileLen=BkFile.tell()
        if BkFileLen == 0L:     #Check if the length of the
                                #file is zero
            print
            print
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print 'xxxxxx NO RECORDS AVAILABLE xxxxxx'
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print
            print
            BkFile.close()
        else:
            BkFile=open('BookDetails', 'r')     #Opens the file
                                                #in read mode
                                                #to print all
                                                #its records
            print
            print
            print '###################################'
            print '########## BOOK DETAILS ##########'
            print '###################################'
            print
            end=0
            record=1
            while not end:
                BkDet=BkFile.readline()
                if BkDet != '':
                    print
                    print 'Record number: %s' % (record)
                    print '================'
                    print BkDet
                    record = record + 1
                else:
                    print
                    print '***********************************'
                    print '********** END OF FILE **********'
                    print '***********************************'
                    print
                    print
                    end=1
                    BkFile.close()

class software(library):        #Defines the library class,
                                #which is a subclass of the
                                #library class
    'software class'
    def __init__(self):
        'software class constructor'
        ProductOf=Size=''       #Initializes the attributes of
                                #the library class

    def sws_method(self):       #Takes input for software
```

```
                                #details
        'Enter software details'
        SwFile=open('SoftwareDetails', 'a')
        libM=self.lib_method()      #Calls the method of the
                                    #base class, which takes
                                    #input for three
                                    #attributes, LibCode,
                                    #Title, and Price, and
                                    #returns their values
        SwFile.write(libM[0] + ',')
        SwFile.write(libM[1] + ',')
        ProductOf=raw_input('Enter the name of the software vendor: ')
        SwFile.write(ProductOf + ',')
        Size=raw_input('Enter the size of the software (in MB): ')
        SwFile.write(Size + ',')
        SwFile.write(libM[2] + '\n')
        SwFile.close()
        print '''
You have entered the following details for a book:
==================================================
Library code: %s
Title: %s
Vendor: %s
Size: %sMB
Price: $%s''' % (libM[0],libM[1],ProductOf,Size,libM[2])

    def sws_display(self):      #Displays all software records
                                #available in the
                                #SoftwareDetails file
        'Display software details'
        SwFile=open('SoftwareDetails', 'a')
        SwFile.seek(0,2)
        SwFileLen=SwFile.tell()
        if SwFileLen == 0L:     #Check if the length of the
                                #file is zero
            print
            print
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print 'xxxxxx NO RECORDS AVAILABLE xxxxxx'
            print 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
            print
            print
            SwFile.close()
        else:
            SwFile=open('SoftwareDetails', 'r')
            print
            print
            print '##################################'
            print '######## SOFTWARE DETAILS ########'
            print '##################################'
            print
            SwFile=open('SoftwareDetails', 'r')
```

```
            end=0
            record=1
            while not end:
                SwDet=SwFile.readline()
                if SwDet != '':
                    print
                    print 'Record number: %s' % (record)
                    print '==============='
                    print SwDet
                    record = record + 1
                else:
                    print
                    print '***********************************'
                    print '********** END OF FILE **********'
                    print '***********************************'
                    print
                    print
                    end=1
            SwFile.close()

def MainMenu():     #Displays the main menu, takes input for
                    #choice, and calls an appropriate method
                    #based on the choice
    MenuItems='''
    <#><#><#><#><#><#><#><#><#>
    <#><#><#><#><#><#><#><#><#>
    <#>                     <#>
    <#> TECHSITY UNIVERSITY <#>
    <#>      LIBRARY        <#>
    <#>                     <#>
    <#><#><#><#><#><#><#><#><#>
    <#><#><#><#><#><#><#><#><#>
            MAIN MENU
            =========
    1 Enter details for books
    2 Enter details for software
    3 View details of books
    4 View details of software
    5 Delete all book records
    6 Delete all software records
    7 Quit
    Enter choice (1-7): '''
    done=0
    while not done:
        MenuChoice=raw_input(MenuItems)     #Asks input for
                                            #choice
        ClearScreen = os.system('clear')
        print 'You entered: %s' % MenuChoice
        if MenuChoice not in '1234567':     #Checks if the
                                            #choice is correct
            print
            print 'Wrong choice. Enter 1, 2, 3, 4, 5, 6, or 7.'
```

```
        print
    else:
        if MenuChoice =='7':        #Quits if the choice is
                                    #7
            done=1
        if MenuChoice =='1':
            print
            print
            print '         ENTER BOOK DETAILS'
            print '         =================='
            print
            bk.bks_method()     #Calls bks_method() of the
                                #books class to accept book
                                #details

            bk.clear_screen_method() #Calls the
                                     #clear_screen_method()
                                     #of the library class
                                     #to clear the screen
        if MenuChoice =='2':
            print
            print
            print '         ENTER SOFTWARE DETAILS'
            print '         ======================'
            print
            sw.sws_method()     #Calls sws_method() of the
                                #software class to accept
                                #software details

            sw.clear_screen_method() #Calls the
                                     #clear_screen_method()
                                     #of the library class
                                     #to clear the screen
        if MenuChoice =='3':
            bk.bks_display()        #Calls bks_display() of
                                    #the books class to
                                    #display all book
                                    #records
            bk.clear_screen_method()
        if MenuChoice =='4':
            sw.sws_display()        #Calls sws_display() of
                                    #the software class to
                                    #display all software
                                    #records
            sw.clear_screen_method()
        if MenuChoice =='5':
            bk.empty_file_method('BookDetails')
            #Calls empty_file_method() of the library
            #class and passes the name of the file to
            #delete all its records

            bk.clear_screen_method()
```

```
            if MenuChoice =='6':
                sw.empty_file_method('SoftwareDetails')
                #Calls empty_file_method() of the library class
                #and passes the name of the file to delete all
                #its records

                sw.clear_screen_method()

bk=books()      #Creates instance of the books class
sw=software()       #Creates instance of the software class
MainMenu()      #Calls the MainMenu() function
```

Execute the Code

To be able to implement or view the output of the code to automate the books and software sections of the Techsity University library, you need to execute the following steps:

1. Write the preceding code in a text editor and save it with the .py extension.
2. At the shell prompt, type python followed by the name of the file if the file is in the current directory.
3. Use the Main menu (see Figure 8.2) to add, view, and delete details about books and software.

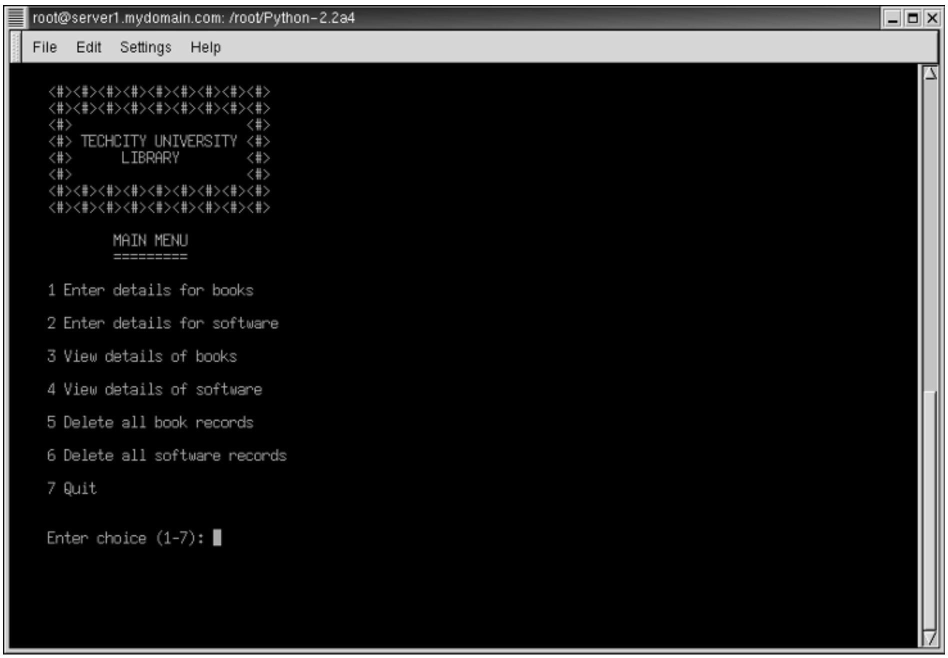


Figure 8.2 The main menu.

Summary

In this chapter, you learned the following:

- The object-oriented approach to programming has changed the way programs are written today.
- Object-oriented programming (OOP) has the following two major components:
  - Objects
  - Classes
- OOP has the following benefits:
  - Models the real world
  - Allows code reusability
  - Is favorable to change
- In Python, all the data types are objects, and the word "object" need not mean an instance of a class.
- Python classes are data structures used to define objects.
- You can work with class objects by performing the following two types of operations:
  - Creating attribute references
  - Creating an instance of a class
- A class attribute is an element of a class.
- The class attributes belong to the class in which they are defined.
- The class attributes are of the following two types:
  - Data attributes
  - Functional attributes
- Data attributes are commonly known as static members or class variables and are set when the class is created.
- Functional attributes or method class attributes are the class methods.
- Methods can be invoked only by using an instance of the class to which they belong.
- A class instance is a variable that contains a reference to a class.
- The process of creating an instance of a class is known as *instantiation*.
- `__init__()` is a constructor or a special method that can be defined in a class to create objects in the initial state.
- The `__init__()` special method has `self` as the first argument like any other function or method defined in Python.

- Classes can be implemented in the following two ways:
  - Composition
  - Derivation
- In composition, classes are combined to create a code that provides better functionality.
- Derivation provides a powerful feature of OOP, which allows for the use of the features and behavior of a class by another class without disturbing the rest of the program.
- The term "subclass" describes a class that inherits or derives the attributes from another class.
- The term "base class" describes a class from which a subclass has been derived.
- Subclasses inherit most of the attributes of their base classes.
- Inheritance is the property by which a subclass derives the attributes of the base class.
- In inheritance, the base class is also termed as the *parent* and the subclass as the *child*.
- When a subclass is inherited from multiple base classes, it is known as *multiple inheritance*.
- There might be times when you use the same names for methods in the base class and the subclasses. In such a situation, the methods in the base classes override the methods of their subclasses. This is known as method overriding.
- Python has some of the following common built-in functions for OOP:
  - `dir()`
  - `var()`
  - `isinstance()`
  - `issubclass()`
  - `hasattr()`
  - `getattr()`
  - `setattr()`
  - `delattr()`
- Python allows you to modify, add, or remove some functionality to an existing object, such as a data type or some code by packaging the object. This is known as wrapping.
- Delegation is a characteristic of wrapping that uses the existing functionality of the type to enable code reusability.

CHAPTER 9

Exception Handling

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Identify basics of exceptions**
- **Identify standard exceptions in Python**
- **Handle exceptions**
- **Raise exceptions**
- **Create user-defined exceptions**

Getting Started

While shopping for fruits and vegetables from a grocery shop, will you buy them if they are rotten? Imagine that, while you are driving, your car stops after every little pebble that comes your way and you have to restart it every time. Similarly, errors in the program execution may cause your program to come to a fatal stop or may produce garbage output. In such a situation, you might have to re-execute the program to show the output or correct the error that caused the problem. It's nice that your car is designed with features that handle little hurdles very well. In the same way, you can also construct your programs to handle possible errors.

Thus, error handling is important to account for unexpected situations, such as insufficient memory or inability to find or open files. If these errors are not trapped, the program can come to an abrupt halt or produce unwanted output. The program can show anomalous behavior because of two types of problems, syntax errors and exceptions. *Syntax errors* are the errors that occur when a statement or a command is not written in the way that is allowed by the software. Thus, syntax errors cannot be compiled by the interpreter and have to be repaired before starting the execution. On the other hand, an *exception* can be defined as the unexpected event that occurs during the execution of a program and disrupts the normal flow of instructions. In most instances, exceptions cause program disruption, and the interpreter reaches a point where it cannot continue the program execution any further. Exceptions are erroneous events like a division by zero or a request for out-of-range index for a sequence.

Most of the time you need the program to complete execution of other parts even if an error occurs in one part. This can be accomplished through exception handling. The exception handling in Python allows programs to handle abnormal and unexpected situations in a structured and ordered manner.

The action as a resolution for exception can occur in two phases. The first phase is the error, which actually causes the exception to occur, and the second phase is where the exception is detected and resolved. Let's elaborate on these phases.

When the error occurs, the Python interpreter tries to identify it. This is called throwing an exception (also known as triggering and generating). Throwing an exception is the process by which the interpreter tells the program control that there has been an anomaly. Python also allows the programmer to raise an exception. Whether user-defined or triggered by the Python interpreter, exceptions indicate that the error has occurred. The appropriate action to resolve the error can be taken in the second phase.

When an exception is raised, a host of possible actions can be invoked in response: ignoring the error and resuming the program flow, logging the error but taking no action, rectifying the problem that caused the exception to occur, or performing another action and aborting the program.

This chapter explains exceptions and the phases in which the actions related to an exception are performed. Next, the chapter introduces you to the standard exceptions in Python. This chapter further explains how exceptions can be raised. Finally, the chapter explains user-defined exceptions.

Handling Exceptions

Problem Statement

Jim, the data analyst, has written a code that accepts student details and displays them after calculating the scholarship applicable for each student. The code, however, generates an error and halts unusually. The code for accepting and displaying student details is given here. Jim now needs to control program execution so that the execution does not terminate abruptly.

```
class Student:
    def __init__(self,name,phno,fee,age=18,schrship=0.15):
        self.studname=name
```

```
        self.studphno=phno
        self.studage=age
        self.studfee=fee
        self.studschrship=schrship
    def displaydetails(self):
        print '%-20s %s' % ('Name:',self.studname)
        print '%-20s %d' % ('Age:',self.studage)
        print '%-20s %s ' % ('Phone number:',self.studphno)
        print '%-20s %f' % ('Course fee:',self.studfee)
        print '%-20s %f' % ('Scholarship(%):',self.studschrship)
        scship=self.studfee-(self.studschrship*100/self.studfee)
        print '%-20s %f' % ('Scholarship($):',scship)
        print '\n'
r=os.system("clear")
studobjects=[]
studobjects.append(Student('Tom','5552383745',4000))
studobjects.append(Student('Mac','6478638323',4500,22))
studobjects.append(Student('Leonard','8485242263',6500,19,0))
ctr=0
while ctr<=3:
    studobjects[ctr].displaydetails()
    ctr=ctr+1
print('All displayed')
```

The error generated by the preceding code, which is saved as **studentdetails.py**, is shown in Figure 9.1.

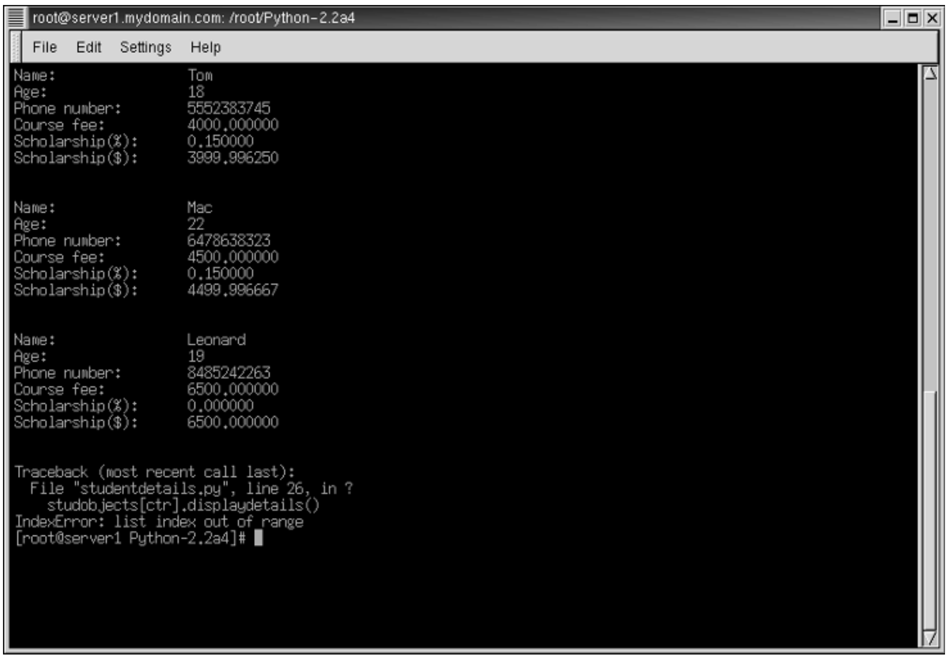


Figure 9.1 The output of studentdetails.py.

Task List

- ✓ **Identify the type of error and where the error occurs.**
- ✓ **Identify the mechanism of trapping the exception.**
- ✓ **Identify the location where the code for handling the exception has to be written.**
- ✓ **Write the code for handling the exception.**
- ✓ **Save and execute the code.**

Identify the Type of Error and Where the Error Occurs

Recall that the earlier chapters in this book had examples that included code snippets in which errors occurred. Whenever the Python interpreter encounters an error, it displays the information related to that error, such as the name of the error, the reason for the error, and, most of the time, the line number where the error occurred. All errors have a similar format whether they occur while running a script or at the Python prompt. As discussed earlier, these errors occur due to the program's anomalous behavior that is incompatible with the Python interpreter. Let's have a look at some of the common exceptions that occur.

`ZeroDivisionError.` This error occurs when any number is divided by numeric zero. For example,

```
>>> 55/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

`NameError.` This error occurs when an attempt is made to access a variable that has not been assigned. `NameError` indicates that the identifier was not found in the interpreter's symbol table. The Python interpreter searches for a variable in the global and local namespace and returns `NameError` if it does not find the variable in any of these namespaces.

```
>>> ruf
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'ruf' is not defined
```

`SyntaxError.` As stated earlier, syntax errors do not occur at run time. When a `SyntaxError` exception is raised, it indicates that a piece of code or a statement is not written according to the syntax allowed in Python. These exceptions occur at compile time and have to be corrected before the execution of the program. For example,

```
>>> def
Traceback (  File "<interactive input>", line 1
    def
       ^
SyntaxError: invalid syntax
```

The preceding command generates an error because the `def` keyword must follow a name of a function. Because the Python interpreter expects an identifier after the `def` keyword, it gives an error.

IOError. This error occurs due to general input/output failures, such as inability to read from a file or attempting to access a nonexistent file. For example,

```
>>> file=open("Myfile")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'Myfile'
```

In the preceding example, the error occurs because the interpreter tries to search for `Myfile` and it cannot find the file.

IndexError. This error is generated when an attempt is made to access an element beyond the index of a sequence. For example, if you try to access the second element of a list that contains only one element, `IndexError` will be thrown as follows:

```
>>> Mylist=['abc']
>>> Mylist[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

KeyError. You know that in a dictionary, values are mapped to a corresponding key. `KeyError` occurs when a request is made to access a nonexistent key in the dictionary. For example,

```
>>> dict1={'name':'mac','ecode':6734,'dept':'sales'}
>>> dict1['telno']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: telno
```

ImportError. This error is generated when an attempt is made to import a module that does not exist or the interpreter is unable to locate it. This error can also occur when the `from-import` statement is not able to import a name that is requested. Following is the example of when the import statement fails.

```
>>> import mod
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named mod
```

Table 9.1 describes the standard exceptions in Python. Prior to Python 1.5, all exceptions were identified as string objects; however, in Python 1.5 and later versions, most exceptions are provided as class objects. The exceptions are defined in the `exceptions` module. You do not need to import the `exceptions` module explicitly. All the exceptions are built in the namespace by default.

Table 9.1 Standard Exception Hierarchy

EXCEPTION NAME	DERIVED FROM	DESCRIPTION
`Exception`		Base class for all exceptions.
`StopIteration`	`Exception`	Raised when the `next()` method of an iterator does not point to any object.
`SystemExit`	`Exception`	Raised by the `sys.exit()` function.
`StandardError`	`Exception`	Base class for all built-in exceptions except `StopIteration` and `SystemExit`.
`ArithmeticError`	`StandardError`	Base class for all errors that occur for numeric calculation.
`OverflowError`	`ArithmeticError`	Raised when a calculation exceeds maximum limit for a numeric type.
`FloatingPointError`	`ArithmeticError`	Raised when a floating point calculation fails.
`ZeroDivisonError`	`ArithmeticError`	Raised when division or modulo by zero takes place for all numeric types.
`AssertionError`	`StandardError`	Raised in case of failure of the Assert statement.
`AttributeError`	`StandardError`	Raised in case of failure of attribute reference or assignment.
`EOFError`	`StandardError`	Raised when there is no input from either the `raw_input()` or `input()` function and the end of file is reached.
`ImportError`	`StandardError`	Raised when an `import` statement fails.
`KeyboardInterrupt`	`StandardError`	Raised when the user interrupts program execution, usually by pressing `Ctrl+c`.
`LookupError`	`StandardError`	Base class for all lookup errors.

EXCEPTION NAME	DERIVED FROM	DESCRIPTION
`IndexError`	`LookupError`	Raised when an index is not found in a sequence.
`KeyError`	`LookupError`	Raised when the specified key is not found in the dictionary.
`NameError`	`StandardError`	Raised when an identifier is not found in the local or global namespace.
`UnboundLocalError`	`NameError`	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
`EnvironmentError`	`StandardError`	Base class for all exceptions that occur outside the Python environment.
`IOError`	`EnvironmentError`	Raised when an input/ output operation fails, such as the `print` statement or the `open()` function when trying to open a file that does not exist.
`OSError`	`EnvironmentError`	Raised for operating system-related errors.
`SyntaxError`	`StandardError`	Raised when there is an error in Python syntax.
`IndentationError`	`SyntaxError`	Raised when indentation is not specified properly.
`SystemError`	`StandardError`	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
`SystemExit`	`StandrdError`	Raised when Python interpreter is quit by using the `sys.exit()` function. If not handled in the code, causes the interpreter to exit.
`TypeError`	`StandardError`	Raised when an operation or function is attempted that is invalid for the specified data type.

continues

Table 9.1 Standard Exception Hierarchy (Continued)

EXCEPTION NAME	DERIVED FROM	DESCRIPTION
`ValueError`	`StandardError`	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
`RuntimeError`	`StandardError`	Raised when a generated error does not fall into any category.
`NotImplementedError`	`RunTimeError`	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Result

The type of error that occurs in the program for displaying student details is `IndexError`. The code segment in which the error occurs is the following statement:

```
studobjects[ctr].displaydetails()
```

Identify the Mechanism of Trapping the Exception

When an unexpected error occurs in the program, the Python interpreter creates an object of the appropriate exception class. As discussed earlier, this is the first phase where, after creating the object, the Python interpreter passes it to the program by throwing the exception. The exception object contains the information about the type of the error and the state of the object when the exception occurred. Then, you can write the code to handle the exception using an *exception handler*. Various exception handling techniques can be used to trap an exception and then give instructions to the interpreter based on the exception that occurs.

Exception-Handling Techniques

The exception handler code can be implemented in a `try` statement. The `try` statement can be implemented in two forms, `try-except` and `try-finally`. Let's discuss each of these in detail.

The `try-except` Statement

The `try-except` statement allows you first to throw an exception in the `try` block and then write the diagnostic code to handle the exception in the `except` block. The syntax of the `try-except` statement is this:

```
try:
   try_statements
except Exception:
   except_statements
```

The `try_statements` block, which is after the `try` statement, contains the statements that define the scope of exception handlers associated with it. The `except_statements` block, after the `except` statement, contains the exception-handler code immediately after the `try_statements` block. The `except` statement catches the specified exception and executes the `except_statements` block. Let's consider an example to understand better how the `try-except` statement works.

```
>>> try:
...   import mod
... except ImportError:
...   print "Cannot locate the module"
...
Cannot locate the module
```

In the preceding example, the attempt to open the module `mod` is made in the block of code below the `try` statement. When the specified module does not exist, the exception occurs. As you can see, the exception still occurs, so what is the use of exception handling? The answer to this question lies in the `except` statement. During program execution the interpreter tries to execute all statements in the `try` block. If no exception occurs, the statements in the `except` block are not executed, and any code after the `try-except`-statement is executed. If an exception occurs that is specified in the `except` statement, the code in the `except` block is executed. If an exception that is not specified in the `except` statement occurs, then the example here does not include the exception-handling code for that exception. In this example, occurrence of any other exception than `ImportError` will cause the program to halt execution. What do you do if another exception occurs? To handle multiple exceptions, you can also write multiple except statements for a single `try` statement or catch multiple exceptions in a single `except` statement. Before elaborating on each of these, let's first discuss the different ways in which an exception can be handled. Let's consider an example to explain this.

NOTE Remember that there should not be any statement between the `try` block and its corresponding `except` block. A `try` block should be immediately followed by an `except` block.

You know that the `int()` function converts a string value containing only alphanumeric characters to an integer. If the string passed as an argument to the `int()` function does not contain alphanumeric characters, it gives `ValueError` as follows:

```
>>> int('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): abc
```

It can also give `TypeError` if an argument other than a string is passed as follows:

```
>>> int([12])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object can't be converted to int
```

Consider a user-defined function `int_convert()` that takes an object as a parameter and contains code to convert the object to an integer.

```
def int_convert(var):
   try:
      return int(var)
   except ValueError:
      pass
```

Notice that the preceding code contains a `try` block for attempting to convert `var` to an integer. The `except` block catches `ValueError` if it occurs but simply ignores it. You can also choose to return a value if the exception occurs so that the function actually always returns a value even if the exception occurs. For example,

```
def int_convert(var):
   try:
      return int(var)
   except ValueError:
      return 0
```

You can also choose to print an appropriate message on the screen or store it in a variable.

```
def int_convert(var):
   try:
      return int(var)
   except ValueError:
      print 'The argument does not contain numbers'
```

Notice that the preceding example handles the `ValueError` exception in different ways but does not handle the `TypeError` exception at all, which might occur if any object other than a string is passed to the function. Another exception that is expected to occur can be handled using the following approaches:

- A `try` statement with multiple `except` statements
- A single `except` statement with multiple exceptions

Let's discuss each of them in detail.

A `try` Statement with Multiple `except` Statements. A single try statement can have multiple `except` statements. This is useful when the `try` block contains

statements that may throw different types of exceptions. The syntax for multiple `except` statements is this:

```
try:
   try_statements
except Exception1:
   except_statements1
except Exception2:
   except_statements2
:
:
except ExceptionN:
   except_statementsN
```

In this form of `try-except` statement, the interpreter attempts to execute the statements in the try block. If an exception is thrown and a match is found in an `except` statement, the corresponding `except_statements` block is executed. Let's come back to our example of the `int_convert` function. The `ValueError` that was expected to occur was handled in one `except` statement; however, `TypeError` was not handled. Let's write another `except` statement to handle `TypeError`.

```
>>>def int_convert(var):
...    try:
...       return int(var)
...    except ValueError:
...       print 'The variable does not contain numbers'
...    except TypeError:
...       print 'Non-string type can\'t be converted to integer'
```

You can execute the preceding code using different function calls as follows:

```
>>>int_convert('abc')
The variable does not contain numbers
>>>int_convert([12])
Non-string type can't be converted to integer
>>> int_convert('12')
12
```

Single `except` Statement with Multiple Exceptions. You can also use the same `except` statement to handle multiple exceptions. This can be a situation when you do not want to perform different actions when any exception occurs. The syntax of the `except` statement with multiple exceptions is this:

```
try:
   try_statements
except (Exception1[,Exception2[,...ExceptionN]]]):
   except_statements
```

When multiple exceptions are handled in a single `except` statement, they are specified as a tuple. Let's change the `int_convert()` function to display the same message when either `ValueError` or `TypeError` occur.

```
>>>def int_convert(var):
...    try:
...       return int(var)
...    except (ValueError,TypeError):
...       print 'Wrong argument type or the argument contains
alphabetic characters'
```

You can execute the preceding code using different function calls as follows:

```
>>>int_convert('abc')
Wrong argument type or the argument contains alphabetic characters
>>> int_convert([12])
Wrong argument type or the argument contains alphabetic characters
>>> int_convert('12')
12
```

NOTE You can also use the `except` statement with no exceptions defined as follows:

```
try:
   try_statements
except:
   except_statements
```

This kind of a `try-except` statement catches all the exceptions that occur. Using this kind of `try-except` statement is not considered a good programming practice, though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

You learned the different ways in which an exception can be handled. The question now arises, what do you do if you also want the `except` statement to return the value of the exception? Let's learn how to return values by using the `except` statement.

Argument of an Exception. An exception may have an associated value, called the *argument* of the exception. Every time an exception is thrown, an instance of the `exception` class is created. The argument of an exception and its type depend on the exception class. If you are writing the code to handle a single exception, you can have a variable follow the name of the exception in the `except` statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception. This variable will receive the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location. Following is an example for a single exception:

```
def int_convert(var):
   try:
      return int(var)
   except ValueError,arg:
      print 'The argument does not contain numbers:',arg
```

When you execute the preceding call using the function call
`int_convert('abc')`,

the output will be:

```
The argument does not contain numbers: invalid literal for int(): abc
```

Notice that, because `ValueError` contains only one value in its argument, the output contains a single value. Following is an example for multiple exceptions:

```
def int_convert(var):
   try:
      return int(var)
   except (ValueError,TypeError), arg:
      print 'Wrong argument type or the argument contains alphabetic
characters:', arg
```

When you execute the preceding call using the function call
`int_convert([12,13])`,

the output will be:

```
Wrong argument type or the argument contains alphabetic characters:
object can't be converted to int
```

After discussing various forms of the `try-except` statement, let's learn how the `else` statement works with the `try-except` statement.

The `else` Statement. There may be some statements that you want to execute if the `try` statement does not generate any errors. One way out is that you can place these statements in the `try` block. You may not always want to do this, though, because these statements might generate some exceptions, which will be caught in the subsequent `except` statements. To solve this problem, you can use the `else` statement. The `else` statement is placed after all the `except` blocks for a particular `try` block and contains code that must be executed when no exception is raised by the `try` statement. For example,

```
def int_convert(var):
    try:
        print int(var)
    except ValueError:
        print 'The variable does not contain numbers'
    except TypeError:
        print 'Non-string type can\'t be converted to integer'
    else:
        print 'No exception generated'
```

In the preceding example, note that the else statement is placed after all the `except` statements. When you call the `int_convert()` function by using the function call `int_convert('32')`, the output will be:

```
32
No exception generated
```

You can also nest `try-except` statements. Nested `try` blocks are similar to nested constructs. You can have one `try` block inside another. Similarly, an

except block can also contain other try-except statements. If the lower-level try-except block does not have a matching except handler, the outer try block is checked for it.

The try-finally Statement

When a statement in the try block causes an exception to occur, the rest of the statements in the try block are ignored. At times those statements must be executed regardless of the occurrence of the exception. You can place all such statements in the finally block. The syntax of the try-finally statement is this:

```
try:
   try_statements
finally:
   finally_statements
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement. Consider the following example:

```
try:
    f=open('testfile','w')
    f.write('Bank calculations for interest')
except IOError:
    print ' Error: can\'t find file or read data'
    f.close()
```

In the preceding example, the statements in the try block are executed. If an exception occurs, it is handled in the except block and the file is closed. What happens when the exception does not occur? The file is never closed and remains open. Shifting the f.close() statement to the try block will also not solve our problem. In that case, when an exception occurs the file will remain open and the program control will exit the try-except statement without closing the file. The finally statement comes to your rescue in situations like these where certain statements need to be executed whether or not the exception occurs. Let's write the preceding code again to demonstrate the use of the try-finally statement.

```
try:
    f=open('testfile.txt','w')
    try:
        f.write('Bank calculations for interest')
    finally:
        f.close()
except IOError:
    print 'Error: can\'t find file or read data'
```

The preceding code uses a nested try-finally statement inside another try-except statement. The outer try block first attempts to open the file. The inner try statement writes a line to the file and immediately jumps to the finally block and

closes the file whether or not an exception occurred while writing to the file. If an exception occurs in the inner `try` block, it is handled in the outer `except` block along with the exception that occurs in the outer `try` block.

Until now, you learned how standard errors raised by the interpreter are generated and trapped. Python also allows you to explicitly generate exceptions. Let's learn how exceptions can be raised.

Raising Exceptions

When raising an exception, the exception can be a Python standard exception or a programmer-defined exception. You can raise exceptions in several ways by using the `raise` statement. The general syntax for the `raise` statement is this:

```
raise [Exception[,argument[,Traceback]]]
```

The first argument of the `raise` statement is the name of the exception to be raised. This name can be the name of a class, a standard exception, or a string. The second argument is optional and contains the arguments for the exception. The previous section has already explained what is an argument of an exception. The third argument, `traceback`, is also an optional argument but is not used too much in practice. A `traceback` object is created when an exception is raised. It is useful when an exception is to be raised again. If not specified, any argument defaults to `None`. Consider an example to raise an exception with an argument as a string.

```
num1=input('Enter num1:')
num2=input('Enter num2:')
op=raw_input('Enter an operator')
if op=='+':
    print num1+num2
else:
    raise ValueError,'Incorrect operator'
```

Running the preceding code will generate an exception with the specified string as the value of the exception.

Any built-in standard exception can be raised by using the `raise` statement. Here, some examples are presented of raising `RuntimeError`.

```
>>> raise RuntimeError
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
RuntimeError
>>> raise RuntimeError()
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
RuntimeError
>>> raise RuntimeError('System not responding')
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
RuntimeError: System not responding
```

You can also raise string exceptions; however, this is not practiced much after class objects were introduced in Python 1.5. The following example illustrates the use of string exceptions:

```
>>> MyError='Incorrect input'
>>> raise MyError
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
Incorrect input
```

Table 9.1 explains that a `KeyboardInterrupt` exception is raised when `Ctrl+c` is pressed or when any of the input functions are waiting for an input from the user. The following code raises a `KeyboardInterrupt` exception and prints a message on the screen.

```
>>> try:
...     raise KeyboardInterrupt
... except KeyboardInterrupt:
...     print 'Sorry u cannot copy'
...
Sorry u cannot copy
```

This section has explained how you can handle built-in standard exceptions. Let's learn about user-defined exceptions.

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions. Here is an example related to `RuntimeError`. Here a class is created that is subclassed from `RuntimeError`. This is useful when you need to display more specific information when an exception is caught. In the `try` block, the user-defined exception is raised and caught in the `except` block. The variable `e` is used to create an instance of the class `Networkerror`.

```
>>> class Networkerror(RuntimeError):
...    def __init__(self,arg):
...       self.args=arg
...
>>> try:
...    raise Networkerror("Bad hostname")
... except Networkerror,e:
...    print e.args
...
Bad hostname
```

After the discussion in this chapter, you will agree that the `try-except` statement will be used to catch the exception that is generated.

Identify the Location for the Code for Handling the Exception to Be Written

Based on the previous discussion, the following result can be obtained for the task of identifying the location where the exception-handling code needs to be written. The error that the code in the problem statement shows is this:

```
IndexError: list index out of range
```

This error occurs in the list used to display the course details, which is part of the `while` loop. Therefore, the `while` loop in the main part of the code has to be enclosed within the `try` block. The exception raised can be caught in the `except` block.

Write the Code for Handling the Exception

Let's now write the code for the problem statement that handles the exception thrown.

```
import os
class Student:
    def __init__(self,name,phno,fee,age=18,schrship=0.15):
        self.studname=name
        self.studphno=phno
        self.studage=age
        self.studfee=fee
        self.studschrship=schrship
    def displaydetails(self):
        print '%-20s %s' % ('Name:',self.studname)
        print '%-20s %d' % ('Age:',self.studage)
        print '%-20s %s ' % ('Phone number:',self.studphno)
        print '%-20s %f' % ('Course fee:',self.studfee)
        print '%-20s %f' % ('Scholarship(%):',self.studschrship)
        scship=self.studfee-(self.studschrship*100/self.studfee)
        print '%-20s %f' % ('Scholarship($):',scship)
        print '\n'
studobjects=[]
studobjects.append(Student('Tom','5552383745',4000))
studobjects.append(Student('Mac','6478638323',4500,22))
studobjects.append(Student('Leonard','8485242263',6500,19,0))
ctr=0
r=os.system("clear")
try: #start of the try block
    while ctr<=3:
        studobjects[ctr].displaydetails()
        ctr=ctr+1
except IndexError: #start of the except block
    print 'Trying to access beyond the length of the list'
else: #Start of statements to be executed
      #if exception does not occur
    print('All displayed')
```

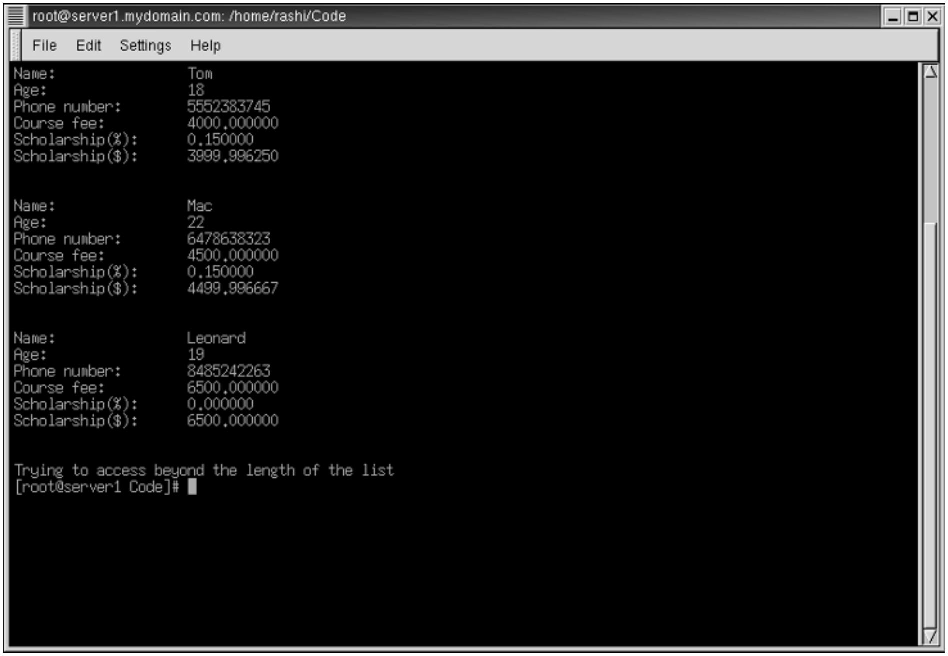


Figure 9.2 Output of the code for exception handling.

Save and Execute the Code

In order to execute the preceding code, do the following:

1. Write the preceding code in a text editor and save it with the `.py` extension.
2. At the shell prompt, type **python** followed by the name of the file if the file is in the current directory.

Figure 9.2 shows the output of the code.

Summary

In this chapter, you learned the following:

- An *exception* can be defined as the unexpected event that occurs during the execution of a program and disrupts the normal flow of instructions.
- The Python interpreter already contains a host of built-in standard exceptions. Whenever the Python interpreter encounters an error, it displays the information related to that error, such as the name of the error, the reason for the error, and, most of the time, the line number where the error occurred.

- You can write the code to handle the exception using an *exception handler*. The exception-handler code can be implemented in a `try` statement.
- The `try` statement can be implemented in two forms:
  - `try-except`
  - `try-finally`
- The syntax of the try-except statement is:

```
try:
   try_statements
except Exception:
   except_statements
```

- The `try` block throws an exception, and the `except` block contains code to handle the exception.
- A single `try` statement can have multiple `except` statements, or an `except` statement can handle multiple exceptions.
- An exception may have an associated value, called the *argument* of the exception. Every time an exception is thrown, an instance of the `exception` class is created. The argument of an exception and its type depend on the exception class.
- The `else` statement is placed after all the `except` blocks for a particular `try` block and contains code that must be executed when no exception is raised by the `try` statement.
- At times those statements must be executed regardless of the occurrence of the exception. You can place all such statements in the `finally` block. The syntax of the `try-finally` statement is this:

```
try:
   try_statements
finally:
   finally_statements
```

- You can raise exceptions in several ways by using the `raise` statement. The general syntax for the `raise` statement is this:

```
raise [Exception[,argument[,Traceback]]]
```

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

CHAPTER

10

CGI Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Describe the various terms and components associated with the Internet**
- **Appreciate the World Wide Web environment**
- **Understand HTTP requests**
- **Use tags and form elements in HTML forms**
- **Differentiate client-side and server-side scripting**
- **Use the `cgi` module**
- **Generate dynamic Web pages by using a CGI application**

Getting Started

Web programming is one of the most important application areas of Python. Python is fast gaining popularity as an Internet programming language. Until now, chapters in this book have introduced you to Python language. These chapters explained the basic concepts of Python and taught you how to write a complete working application by

using Python. They also used a scenario related to a Web application to explain the concepts referred to in a chapter. In this chapter, you will actually delve into the development of Web-based applications in Python. Web programming in Python is performed through CGI scripts.

This chapter assumes that the reader understands basic Internet concepts, such as its working, various types of networks, the client/server architecture, the addressing scheme on the Internet, World Wide Web, and the HTTP request. This chapter also assumes that you know how to create Web pages and forms using HTML. For those of you who are new to these topics, the concepts are discussed briefly in the chapter. Before getting down to writing CGI scripts in Python, let's review the Internet and HTML concepts.

Internet Basics

The origin of the Internet was a result of man's continuous struggle to satisfy human need for fast communication. The first step toward the Internet was connecting two stand-alone computers, which gave birth to local area networks (LANs) and wide area networks (WANs). On these networks, we can quickly share computer equipment, programs, messages, and the information available on a site. Further development in these areas gave birth to a network of many LANs and WANs—the Internet. Figure 10.1 explains how the Internet has made this world a small place. It shows how the Internet connects many other networks. Some of them are NASA, BITNET, NSFNET, ARPANET, and so on.

After this brief introduction to the Internet, let's discuss how the Internet works.

How Does the Internet Work?

One of the key aspects of communication between computers over a network is the transfer of data. This type of communication requires the following:

- The address of the destination
- A safe method of transmitting data in the form of electronic signals

Before we proceed further with the explanation of these two requirements, let's first recap some more terms commonly used in the Internet scenario: the client, the server, and the client/server network. These three terms lay the foundation for understanding how data is transferred over a network.

Client

A *client* is a destination computer on the network that requests services from another computer on the network. This computer requires adequate access permissions to be able to request services and access resources from other computers.

Server

A *server* is a source computer that receives requests from the client computer, processes these requests, and serves the requested information/data to the client computer. The server computer has a range of services to offer to a client; for example, a server

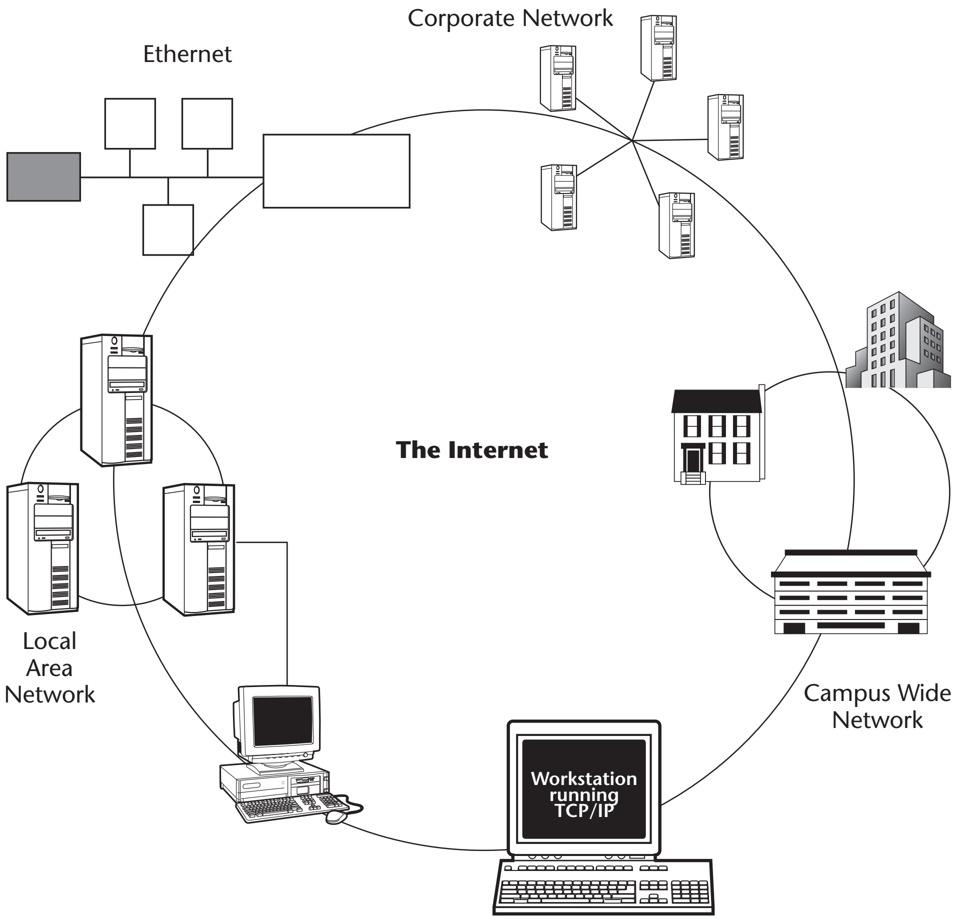


Figure 10.1 The Internet—the network of networks.

computer can offer information, software, games, music, and print services. The client can access these services only if it has adequate permissions. The server computer delineates these permissions for the client.

The Client/Server Network

The *client/server network* forms the basis of computer connectivity on a network. This network consists of several client computers that are connected to the server and also to each other. Let's discuss the request/response cycle in a client/server network.

The client computer sends a request to the server computer. The server computer accepts the request if the client has necessary permissions. Assuming that the server computer accepts the client request, the server then serves the requested information to the client computer.

Figure 10.2 illustrates client computers interacting with server computers in the client/server network.

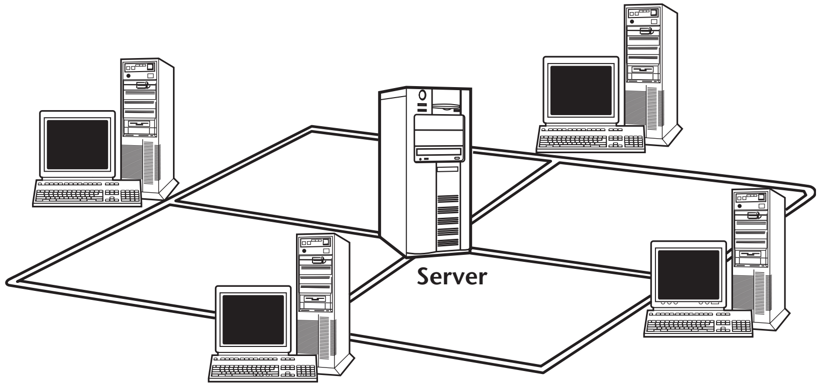


Figure 10.2 Client/server network.

The Internet also follows the client/server architecture where several clients and servers interoperate with each other. In the Internet scenario, a server is also termed a Web server or the host computer, which provides Web services to the clients on the Internet. Not only a server but also a client can host information to another computer, and thus act as a client and a server.

Mode of Data Transmission

Both of the requirements of assigning an address to the destination and providing a method of transmitting data can be taken care of by a set of rules that govern the sending and receiving of data over a network called *protocols*. Some examples of network protocols are TCP/IP, UDP, Apple Talk, and Net BEUI. The Internet uses the TCP/IP protocol to transfer data. The two rules stated previously are implemented in two parts; the first part is called Transmission Control Protocol (TCP), and the second is called Internet Protocol (IP). First, the data to be transferred is divided into small parts called data packets. All the information related to these data packets is also packed with them so that they can be reassembled correctly at the destination without causing any damage to the data. IP assigns a destination address to data packets so that they actually reach the location they are transferred to.

It is not always necessary that all data packets follow the same path from the source to the destination. A special device called the router achieves a balance between the various paths that exist on the Internet. Another computer called the gateway allows different electronic networks to communicate with the Internet, which uses TCP/IP.

Note: In this chapter, we will not delve into the details of the functioning of TCP/IP. The naming and addressing scheme used on the Internet is a mandatory feature for accessing Internet resources. Therefore, let's understand its conventions and abbreviations.

You know that you need a computer, a telephone line or a leased line, and the services of an Internet service provider (ISP) to connect to the Internet. Once you are

connected, volumes of information on varied topics, such as technological guides, topic-specific data, and games, are available at the click of a mouse button. Bear in mind that this information is stored in the form of documents spread over thousands of computers all over the Internet. How are these documents linked so that they are accessible on the Internet? They use the World Wide Web (WWW), an Internet service, as the architectural framework for accessing and linking documents.

World Wide Web

WWW is a common set of protocols that provides standards for specific computers to distribute documents on the Internet. As a result, the World Wide Web is composed of millions of information-holding documents or Web pages distributed on a server, popularly known as a Web server.

A *Web page* is a document created in HTML that includes text, graphics, hypertext links, and audio files. Hypertext Markup Language (HTML) was created as a subset of Standard Generalized Markup Language (SGML) to serve documents over the Internet. HTML uses a simple textual format to create Web pages and contains commands in the form of tags. These tags specify the display format of the various elements of a document. A collection of Web pages about a specific individual or group is known as a Web site. Web sites are created to tender organizational information, advertise for small businesses, offer information, and much more.

NOTE To make a Web page on a site available to everyone, you need to publish the site on a Web server, a process popularly known as Web hosting. Depending on the available financial resources and the size of your Web site, you can choose any of the following methods for publishing a site:

- **You can use your own financial resources or the support of a strong financial partner or institution.**
- **You can use the services of your ISP. Most ISPs allocate some space to dedicated clients for a nominal fee.**
- **You can hire a Web hosting service company to rent you Web space at reasonable rates.**

The documents on the server are accessed through computers that use different platforms, such as Unix, Windows, or Mac OS. As a result, applications at the client end use certain programs to facilitate the display of Web pages in a standardized format. Let's look at a new concept of Web browsers that facilitates a consistent display of Web pages.

Web Browsers

Web browsers are programs that communicate with the Web servers on the Internet and enable the download and display of requested Web pages. Functionally, a Web browser interacts with both the Web server and the operating system of a computer. As a result,

the basic features of a Web browser call for a minimal understanding of HTML and the ability to display text. In recent years, with the advancement of the components of a Web page, expectations from a Web browser have increased multifold. As a result, a Web browser today is able to provide support to complex Web pages with graphics, sound, video, and 3-D imaging.

The most popular Web browsers that have the maximum user support are Netscape Navigator and Microsoft Internet Explorer. Netscape supports a wide range of platforms, such as Windows, Macintosh, and Unix. Internet Explorer, in contrast, was originally designed for Microsoft products but is also available for Macintosh and some Unix platforms today.

The display of the contents of a Web page depends solely on the choice of browser. Therefore, Web applications need to support a standardized display of contents, regardless of the browser used. As a result, an important feature of applications is *cross-browser support*. Cross-browser support ensures a uniform display of content independent of the browser or platform used. This feature of Web applications enables you to view pages in the correct format because of Web applications' compatibility with both browsers, Netscape Navigator and Microsoft Internet Explorer.

Because a browser runs on a client computer, it should contain components that make the display of the Web page contents easy. These components are part of the browser window and are consistent in layout, regardless of their brand. Let's look at the elements of a browser window.

Components of a Browser Window

The basic elements of a Web browser consist of the menu bar, toolbars, the address bar, the viewing window, and the status window. Most of you are familiar with universal elements, such as the menu bar, the viewing window, and the status bar, which are common to nearly all computer applications. The address bar is a browser-specific element that is used to specify the URL of the Web page. The URL contains the name and address of the requested Web page. Figure 10.3 depicts the contents of a Web page.

Uniform Resource Locator (URL)

A *URL*, also called an IP address, contains the exact location of any document. It is an addressing scheme (also called an IP address) that provides the path to an Internet resource. When a user clicks on a link, the URL provides information about that link to the Web browser, which in turn displays the linked Web page. Therefore, links are always implemented by using URLs. A URL may point to a document, image, video, or graphic.

A typical URL is of the following format:

```
protocol://host.domain-name.toplevel-domain-name/path/dataname
```

where:

- *protocol* refers to the type of protocol to be used.
- *host* refers to the server where the resource is stored.



Figure 10.3 A Web page in Microsoft Internet Explorer.

- *domain-name* and *toplevel-domain-name* are the name and the type of the domain, respectively. The types of domains include *com* (used for commercial institutes), *edu* (educational institutes), *net* (network organizations), *org* (miscellaneous organizations), *gov* (government entities), *mil* (US military), *.info* (content sites), *.name* (personal Web sites), and *.biz* (business organizations).
- *path/dataname* refers to the location on the server on which the data is stored.

We will discuss the HTTP protocol in later sections of the chapter.

How does a Web browser use a URL to access HTML documents? The process of accessing HTML documents that represent Web pages consists of the following three steps:

1. The browser determines the protocol to be used.
2. The URL is used to contact the server.
3. The path name and the filename are used to request the specific Web page from the server.

We now know that a client computer uses a browser to display the content of a Web page. We also know that when a Web page is hosted on a server, the client calls for a

particular Web page on the server. When the Web server receives the client call, it responds by displaying the contents of the requested Web page. Can you visualize an ongoing interaction between the client and the server?

In the case of Web applications, Hypertext Transfer Protocol (HTTP) is used to facilitate the exchange of information and data between the client and the server.

Hypertext Transfer Protocol (HTTP)

HTTP is based on the request-response phenomenon in which the client, which is represented by the browser, sends a request to the server, which is represented by the Web server. A typical HTTP transaction between a Web browser and a Web server will take place in the following manner:

1. A TCP/IP connection is established between the client (browser) and the server.
2. The browser sends a request for a particular HTML page.
3. The server locates the file and sends a response in the form of the text content of the requested page.
4. The TCP/IP connection is closed.

Figure 10.4 depicts the interaction between a client and a server.

HTTP uses explicit methods and specifications to structure both request and response messages. Let's now understand the specifications used by a client and a server during an HTTP request-response cycle.

The HTTP Request

An *HTTP request* is sent to the server along with the URL of the requested page, which is typed in the address location bar of the browser. The standard methods of HTTP 1.1 that are used to specify the type of user request are `GET`, `POST`, `HEAD`, `OPTIONS`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`. Of these, with the CGI perspective, only the first two are generally used.

The `GET` Method

This is the simplest and most frequently used request method to request a static resource with inert contents, such as an HTML page. This method is simple because typing the URL of the requested Web page while surfing the Net invokes the `GET`

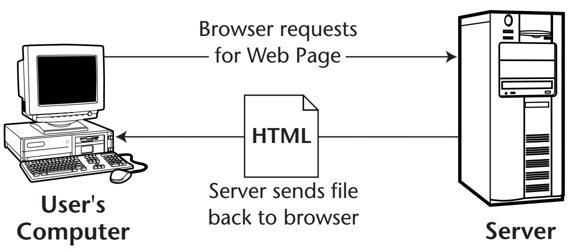


Figure 10.4 The HTTP request-response cycle.

method. As a result, a statement, such as `http://www.mysportspage/index.htm`, specifies the request made using the `GET` method to fetch the specified page contents from the Web server. In addition to making a page request, the `GET` method can be used to include additional information on the Web page. Such additional information is passed as a query parameter that is appended to the URL of the Web page. For instance, at any Web shopping site, have you ever noticed the URL string that appears when you send your logon information for validation? Or, better still, have you noticed the change in the URL string on the status bar when you open your mailbox at the end of the day? The URL address is appended with a string set apart with a "?". The URL is something like this:

```
http://www.URLAddress.com?login=yourLoginName
```

The query parameter in such a case serves as a dynamic search criterion that is used to send parameter-specific content.

The `POST` Method

The `POST` method is used to request a dynamic resource that requires sending large amounts of data as request parameters for the server. Unlike the parameters in the `GET` method, the parameters in this method are contained within the body of the request. Because the size of a request parameter can contain any amount of text, the `POST` method can be used to upload even huge binary and text files. An advantage of using the `GET` method, despite the restriction in the size of the parameter, is that such requests can be used as bookmarks, which can be saved for visits to the same sites in the future.

Before you begin writing programs in Python, it is a good idea to reexamine the basics of HTML. This will help you embed Python code in HTML more effectively.

Revising HTML

You can use HTML along with Python to create attractive and dynamic Web pages. Here are a few pointers to HTML to refresh your memory.

- Hypertext Markup Language (HTML) is the most common markup language that has been used extensively over a period of years to create Web pages. HTML was derived from *Standard Generalized Markup Language (SGML)*; however, HTML is much simpler and easier to use than SGML.
- Markup languages, such as HTML, use *labels*. These labels are used to specify text or images. These labels are called *tags*.
- Tags are used to contain specific *elements* of HTML. Elements are the heart of any HTML document. Elements are nothing but logical blocks that determine how text or an image will appear when displayed on a Web page.
- To use elements, you need to specify them within opening and closing tags. An opening tag marks the beginning of an element, and a closing tag marks the end of an element.
- Each HTML document will always contain the `<HTML> </HTML>` element. The `HTML` element indicates that the text in the document is hypertext.

- Each element has certain characteristics. These characteristics are called the *attributes* of an element. For example, the attributes of the element `<TABLE>` are `ALIGN`, `WIDTH`, `BORDER`, `CELLSPACING`, and `CELLPADDING`. Each of these attributes can be specified within the `TABLE` element.

Consider the following HTML code:

```
<HTML><HEAD><TITLE>Earnest Bank</TITLE>
</HEAD>
<BODY bgColor=#CCCCFF text="#9900FF">
<FORM>
<P><font size="7" color="#9900FF">Earnest Bank<br>
</font><FONT color=#ff6699 face=Arial size=4>Registration form for new
account </FONT></P>
<HR>
<P><FONT color=#cc0099>Please tell us about yourself:</FONT></P>
<P><font color="#9900FF" size="3">Title
(Optional)          &n
bsp
;  
Mr <INPUT CHECKED name=R1 type=radio value=V1>   Mrs
<INPUT name=R1 type=radio value=V2>
Ms <INPUT name=R1 type=radio value=V3>  Dr <INPUT name=R1
type=radio value=V4></font></P>
<P><font color="#9900FF" size="3">First
Name:         
         
<INPUT
name=T1 size=30>  </font>      
     
<font color="#9900FF"><font size="3">Last
Name:          
        
</font><INPUT
name=T2 size=30></font></P>
<P><font color="#9900FF"><font size="3">Social Security
#:         
</font><INPUT name=T3
size=30></font></P>
<P><font color="#9900FF" size="3"><b>Home Address</b></font></P>
<P><font color="#9900FF"
size="3">Apt/Suite#:      
            
<INPUT name=T4 size=30></font>
         
    
<font color="#9900FF"
size="3">Street#:        
          
      
<INPUT name=T5 size=30>
```

```
</font></P>
<P><font color="#9900FF"
size="3">City:        
          
           
<INPUT name=T6
size=30></font>        
    
<font color="#9900FF"
size="3">State:        
       
<INPUT name=T7 size=14> 
Zip:           
<INPUT
name=T8 size=13></font></P>
<P><font color="#9900FF"><font size="3">Home
Phone:           
    
</font><INPUT
name=T9
size=30></font>         
   
<font color="#9900FF" size="3">Email
Id:          
            
</font><INPUT name=T10 size=30></P>
<P><FONT color=#cc0099>Please provide us with your financial
information:</FONT></P>
<P><font color="#9900FF" size="3">
Annual
Income:          
<INPUT
name=T11></font>        
           
            
<font color="#9900FF" size="3">Source of
Income:      </font><SELECT name=D1
size=1> <OPTION selected>Business</OPTION> <OPTION>Service</OPTION>
  <OPTION>Agriculture</OPTION> <OPTION>Other
sources</OPTION></SELECT></P>
<P><FONT color=#cc0099>Please tell us about the account set
up:</FONT></P>
<P><font color="#9900FF" size="3">
Account
Type:          
</font><SELECT
name=D2 size=1> <OPTION selected>Savings account</OPTION> <OPTION>Loan
  account</OPTION> <OPTION>Fixed deposits</OPTION> <OPTION>Recurring
  deposits</OPTION></SELECT></P>
<P> </P>
<P>  
```

```
           
           
         
<INPUT name=B1 type=submit value="Submit Form">  
<INPUT name=B2 type=reset value=Reset></P></FORM>
<P> </P></BODY></HTML>
```

The output of the preceding code appears in Internet Explorer on Windows as shown in Figure 10.5.

The output of the preceding code appears in Netscape Navigator on Linux as shown in Figure 10.6.

In the preceding code, the elements that have been used are these:

`HTML.` This element contains an entire HTML document. In simpler words, this document marks the beginning and end of an HTML document.

`HEAD.` This element is used to specify the header information of the document.

`TITLE.` This element is used to specify the title of the document.

`BODY.` This element contains the body text of the HTML document.

`FONT.` This element is used to alter the font size and color of text.

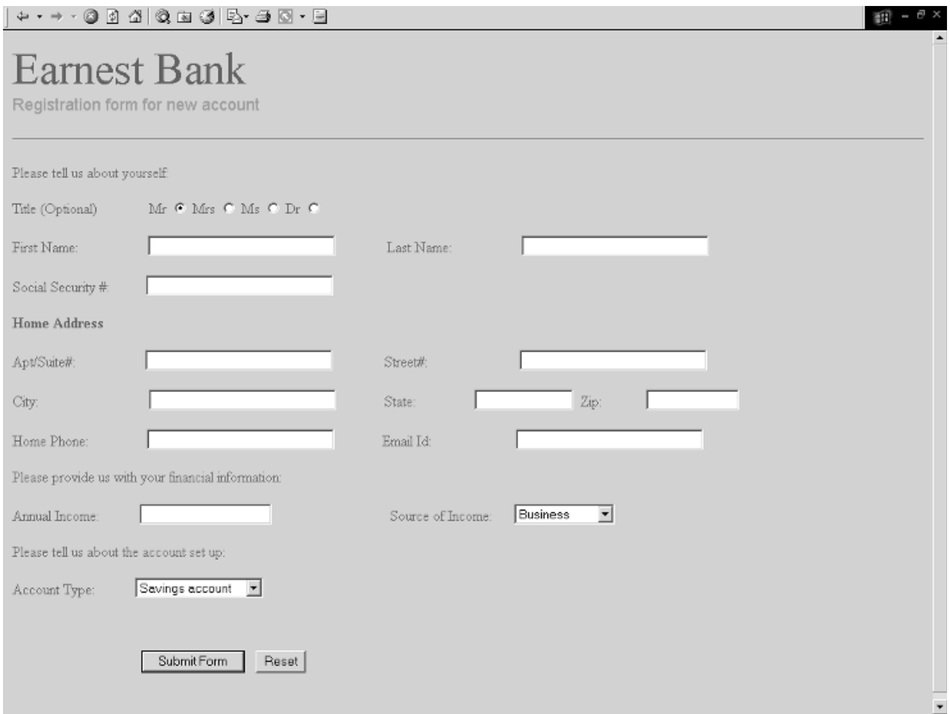


Figure 10.5 Output of sample.html in Internet Explorer.

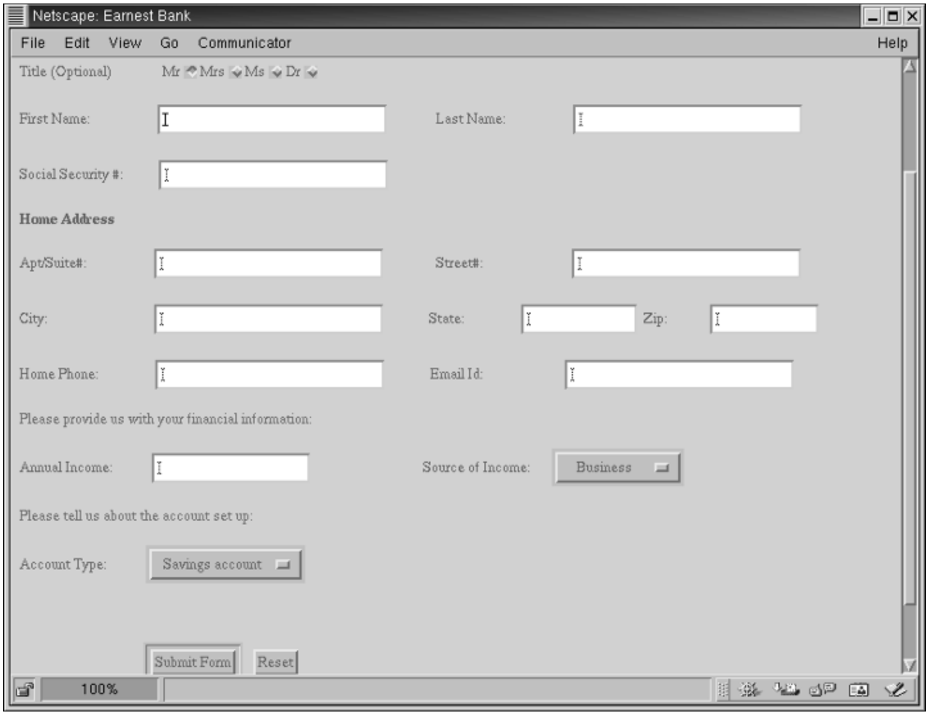


Figure 10.6 Output of sample.html in Netscape Navigator.

`H1.` This element is used to contain that part of the text within the HTML document that needs to appear as Heading 1 on a Web page.

`P.` This element is used to define a paragraph.

`TABLE.` This element is used to define a table that consists of data represented in rows and columns.

`TD.` This element is used to specify the data that will be present in a cell.

`TR.` This element is used to specify TD and TH elements.

`CENTER.` This element indicates that the text should be centered.

`B.` This element stands for Bold. It is used to make the text appear bold.

The preceding code also consists of attributes that are used with elements. The attributes are as follows:

- The `COLOR` attribute is used with the `FONT` element to specify the color of the font.
- The `SIZE` attribute is used with the `FONT` element to specify the size of the text.

You can also design HTML forms to accept data from a user. Regardless of whether you are creating a simple login page or a complex shopping cart, three elements will be used generally. The following three basic elements are detailed in this section:

The `FORM` element. This element contains all the code related to a form. In simpler words, the FORM element contains all the tags that are specific to a form.

The `INPUT` element. This element specifies the code used to create the form controls that accept user input. The INPUT element can contain text boxes, buttons, check boxes, or radio buttons.

The `SELECT` element. This element is used to display lists in a form.

Now, let's discuss each of these in detail.

The `FORM` Element

The `FORM` element contains the entire code specific to a form. A form is a collection of text boxes, radio buttons, check boxes, and buttons. The main purpose of a form in a Web page is to accept user input in a systematic and structured manner.

The `FORM` element consists of all the code used to display text boxes, buttons, or a list of options. Therefore, `INPUT` and `SELECT` elements are also included in the `FORM` element. Two attributes are used with the form element:

- The `METHOD` attribute
- The `ACTION` attribute

Now, let's consider each of these attributes in isolation and understand how they are used.

The `METHOD` Attribute

The `METHOD` attribute is used to transmit form data, which is filled in by the user. Two methods can be used to transmit form data:

- The `GET` method
- The `POST` method

These methods have already been discussed earlier in "The HTTP Request" section.

The `ACTION` Attribute

The `ACTION` attribute is used to specify the target where form data is to be transmitted. Typically, the target is a file that contains the code for processing form data. After processing form data, the file generates the desired output and displays it.

Syntax for `METHOD` and `ACTION` Attributes

Almost every HTML form that accepts user input would typically begin with a `FORM` tag that contains the `METHOD` and `ACTION` attributes. The syntax for using these attributes is this:

```
<form METHOD= "GET/POST" ACTION= "name_of_the_target_file">
```

In the preceding syntax, the METHOD attribute specifies the method of transmission to be used. You can use either the GET method or the POST method with the METHOD attribute. The ACTION attribute specifies the name of the file to which form data will be transmitted.

The INPUT Element

As discussed earlier, the INPUT element is specified within the FORM element. The main purpose of using the INPUT element is to accept user-specific input. The INPUT element helps developers create text boxes, buttons, check boxes, and radio buttons in their forms. This, in turn, makes the Web page interactive and user friendly. All a user has to do is fill out the required fields of a form and click a button to submit the information. The features of the INPUT element can be summed up as follows:

The INPUT element consists of controls, such as text boxes, buttons, radio buttons, and check boxes. Each of these controls contains its attributes. These attributes are the following:

The TYPE attribute. This attribute is used to specify the type of control that will be used to accept input from the user.

The NAME attribute. This attribute is used to specify a name for a control. This name is used to identify a particular control in the form.

The VALUE attribute. This attribute holds the value entered by a user or the default value for a particular control.

While using the INPUT element with HTML forms, you can create five types of controls that accentuate the user interface:

- Submit button
- Text boxes
- Radio buttons
- Check boxes
- Combo boxes

HTML has changed the way data is exchanged over the Internet; however, HTML alone can be used to display only static contents. As discussed in the previous section, a browser requests an HTML file from the Web server by using HTTP. The Web server processes the request by sending the appropriate HTML file, and finally, the browser displays the file to the user. To reflect changes, page contents have to be modified and displayed dynamically. Dynamic content can be displayed on a Web page by using client-side and server-side scripting.

Client-Side versus Server-Side Scripting

The development of Web servers has led to a considerable rise in the need for displaying dynamic content. In client-side scripting, scripts are processed by a browser, whereas in server-side scripting, scripts are processed by a server. In other words, when a browser asks a Web server for an HTML file that contains a client-side script,

the client browser processes the file. This enhances the speed with which the requests are processed because the server is not overloaded with processing the script of every client. This saves a lot of time and allows the server to handle the requests of many more clients at the same time. This distribution of work helps in optimizing the performance of the Web server.

Certain tasks need to be processed only by the server and cannot be handled by client-side scripts. Consider that you need to display the current time of the system on which a Web site is hosted. If you use a client-side script, then each of the browsers requesting the script will display the current time of the machine on which the browser is located. The required result can be obtained only if you use a server-side script. Figure 10.7 illustrates the use of client-side and server-side scripts to display the current time of the server. When the `time()` function of the server in New York is invoked using a client-side script, client browsers in Atlanta, Denver, and Seattle show different times as the current time in New York. When a server-side script is used for the same purpose, the client browsers show the correct time in New York.

Server-side scripting is used when there is a need to develop active Web sites that can interact with databases and allow the customization of the content of a Web page for each user. The benefits of server-side scripting can be listed as follows:

- Server-side scripting allows database interactivity with Web pages.
- Server-side scripting allows the use of templates for creating HTML documents. Templates are files containing the HTML code to which contents from a text file, a database, and other data sources can be retrieved dynamically before displaying the Web page to the user. This allows the information to be changed dynamically instead of changing it manually every time it changes.

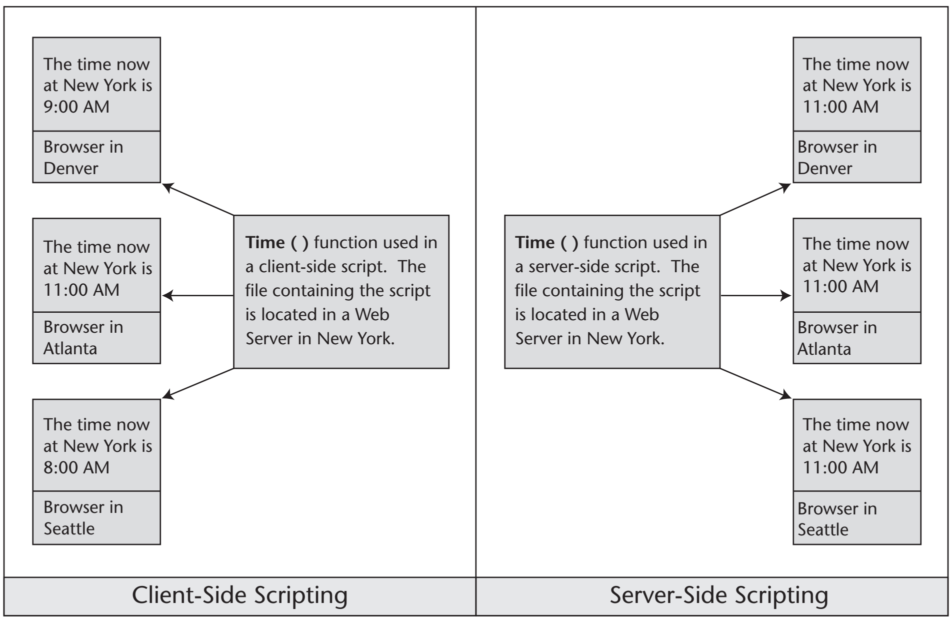


Figure 10.7 Client-side versus server-side scripting.

Python is a powerful server-side scripting language. As stated earlier in this chapter, Web programming in Python is done through CGI. Let's start with an introduction to CGI.

An Introduction to CGI

When a client sends a request to a server by clicking the `Submit` button on an HTML form, the Web server handles the requests in an HTML form by invoking an external program. Both the client and the server wait for the resulting HTML file. After the execution of the external program is completed, the program passes the resulting HTML page back to the server. The server, in turn, passes it to the client. This mechanism of the server receiving the form, contacting the external program to process the request, and receiving and returning the newly generated HTML file is called Common Gateway Interface (CGI). The external program that processes the client request is called a CGI script. Therefore, when a CGI script begins to execute, it also retrieves the data that the user has supplied in an HTML form. This data is supplied on the client browser and does not reside on the server. In other words, the main purpose of CGI is to manage the communication between the client browser and the server. Figure 10.8 explains the working of CGI.

CGI scripts can be written to handle a variety of tasks, such as interaction with databases, files, and other programs on the server and printing the result back to the client in a customized format. CGI scripting can be done in many languages, such as Ruby, ColdFusion, Python, and PHP. Let's write a simple CGI script in Python.

```
#!/usr/local/bin/python
print "Content-Type: text/plain\n\n"
print "Python works"
```

Let's look at each line of the code sequentially. You already know that the first line is the comment to indicate the path to the Python interpreter in a Unix machine. The second line passes the MIME type to the browser and tells the browser how to render the information. This line is important because the browser can understand only HTTP data, which includes HTML and MIME headers. The third line prints the specified line in the browser window. You can write the script in any text editor; however, you have to make sure that you save the file in the `cgi-bin` directory. The complete path to this directory is `/var/www/cgi-bin`. Figure 10.9 shows the output of the preceding script in a browser.

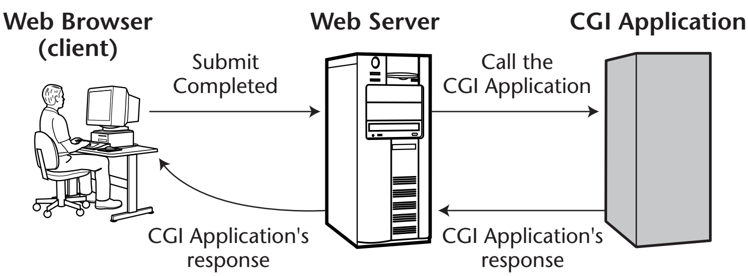


Figure 10.8 The working of CGI.

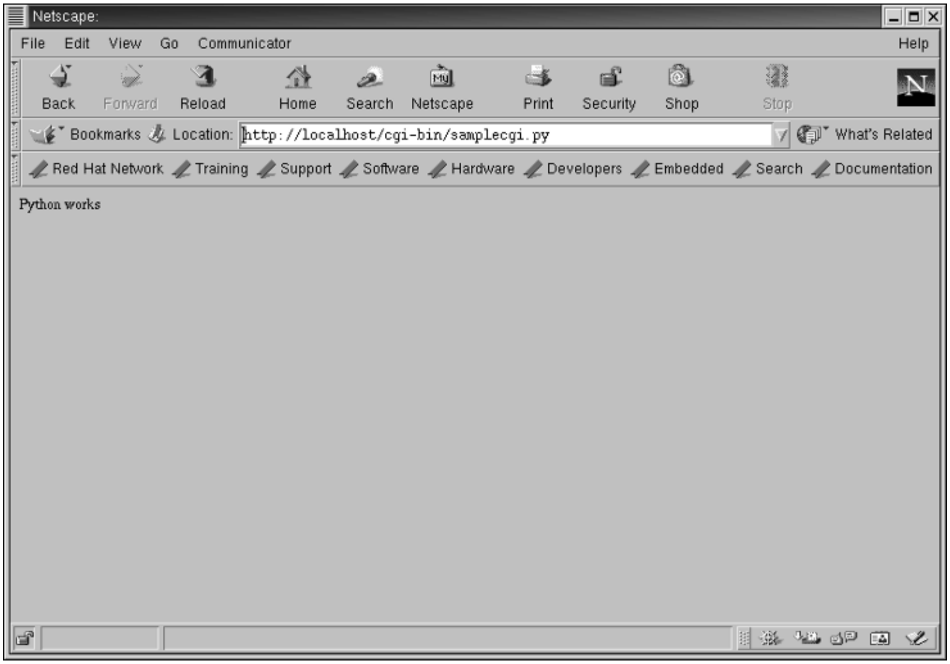


Figure 10.9 Output of the simple CGI script in the browser.

NOTE Often, a CGI script has to be made executable by executing the following command:

```
$ chmod +x scriptname.py
```

Equipped with the basic knowledge of CGI, let's write a complete CGI application in Python.

The `cgi` Module

The `cgi` module has to be imported in any CGI script written in Python. The `fieldstorage` class in the `cgi` module is responsible for communication with a client. When a user enters the data in the client browser, an instance created for the `fieldstorage` class reads standard input from the user in the form of standard input for `POST` calls and a query string for `GET` calls. This instance consists of an object similar to a dictionary in which keys are the names of form items and values are the data in them that was passed through the form. After acquiring the basic knowledge required to write a CGI script, let's write a CGI application for the Techsity University.

Write the CGI Program in Python to Generate the Results Page

As discussed earlier, an HTML page is a static page. After it is created, the contents of the page cannot be changed. When a user sends a request to the server by using an HTML form, the server has to display the results back to the client browser in the form of an HTML page. Therefore, the CGI application that processes the client request should be able to send the results back to the server in the form of a Web page. For this purpose, the CGI application should contain the code to generate an HTML page dynamically. The following section explains how you can write a CGI script by using Python to generate a dynamic Web page.

Generating a Dynamic Web Page

Let's consider an example to explain how data from a form is passed to a CGI script. The example here refers to two files, `details.html` and `results.py`. The following code represents `details.html`, which contains a form to accept the login name and password of a user.

```
<HTML><HEAD><TITLE>
Student Details Form
</TITLE></HEAD>
<BODY>
<b><font size="5"><u>Personal Details Form</u></font></b>
<FORM method="POST" ACTION="http://localhost/cgi-bin/results.py">
<p>Title:
<INPUT TYPE=radio NAME=studtitle VALUE="Mr." CHECKED> Mr
<INPUT TYPE=radio NAME=studtitle VALUE="Mrs."> Mrs.
<INPUT TYPE=radio NAME=studtitle VALUE="Ms."> Ms.
<INPUT TYPE=radio NAME=studtitle VALUE="Dr."> Dr.
<p>Name:
<INPUT TYPE=text NAME=studname VALUE="" SIZE=30></p>
<p>Date of Birth:
<INPUT TYPE=text NAME=studdob VALUE="" SIZE=30></p>
<p>Address:
<textarea NAME=studadd rows=2 cols=30></textarea></p>
<p>Home phone #:
<INPUT TYPE=text NAME=studphone VALUE="" SIZE=30></p>
<p>E-mail address:
<INPUT TYPE=text NAME=emailadd VALUE="" SIZE=30></p>
<P>Course:
<SELECT name=studcourse size=1> <OPTION selected>Project
Management</OPTION>
<OPTION>Quality Management</OPTION>
<OPTION>Team Building</OPTION>
<OPTION>Cost Management</OPTION></SELECT></P>
<INPUT TYPE=submit>
<INPUT TYPE=RESET></FORM></BODY></HTML>
```

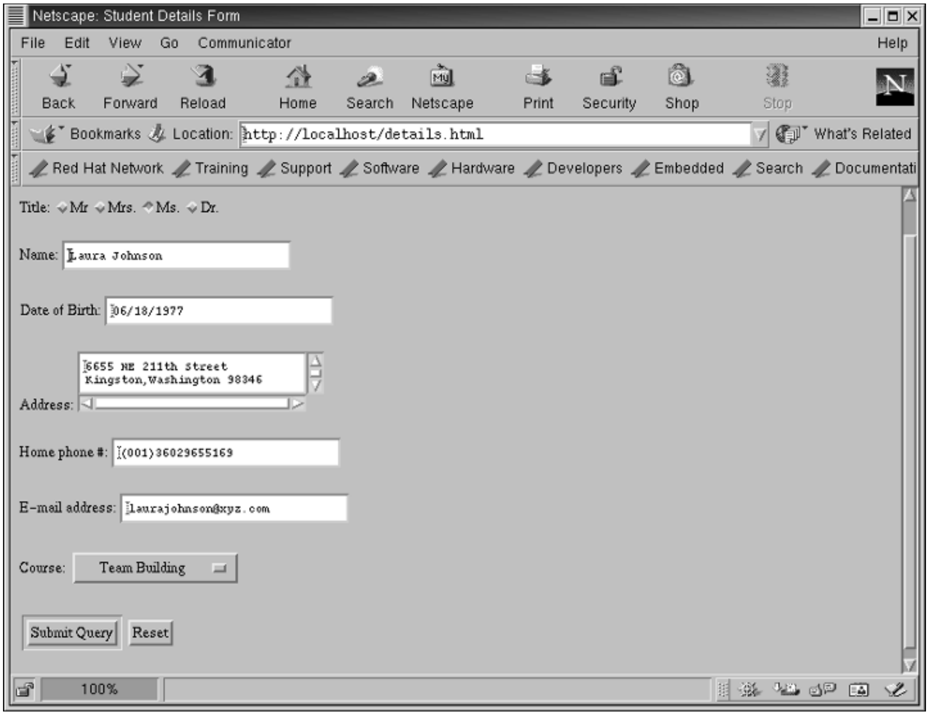


Figure 10.10 Details.html in the Web browser.

The form contains seven data fields: Title, Name, Address, Date of birth, Home phone, E-mail address, and Course. These fields implement radio buttons, text boxes, and a combo box. The form action specifies the type `POST` for the `METHOD` subtag. This means that the data in the HTML form will be parsed to the CGI script `results.py` by using the `POST` method. The path specified is `http://localhost/cgi-bin/results.py` because the CGI script `results.py` is stored in the `cgi-bin` directory. If the `METHOD` subtag is not specified, its default type is assumed to be `GET`. The `POST` method is chosen here because the voluminous data is to be transferred from the HTML form to the CGI script. Figure 10.10 shows the user details page in the browser.

Notice that the address bar of the browser window shows the path of the Web page.

NOTE **By default, the Linux server is configured to run only the scripts in the `cgi-bin` directory in `/var/www`. If you want to specify any other directory to run your CGI scripts, comment the following line in the `httpd.conf` file:**

```
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

and type the following command:

```
<Directory "path_to_httpd_docs">
    Options All
</Directory>
```

Replace "`path_to_httpd_docs`" with the path of the directory where you want to keep the CGI scripts.

When the user clicks the SUBMIT button, the script results.py is executed through CGI. The following is the code for the results.py script, which accepts the user details fields from details.html and creates a dynamic HTML page to show the results.

```
#!/usr/local/bin/python
import cgi
print "Content-Type: text/html\n"
dynhtml='''<HTML><HEAD><TITLE>
Personal Details</TITLE></HEAD>
<BODY><H2>Personal details for: %s %s</H2>
<p>Your date of birth is: <b>%s</b></p>
<p>Your home address is: <b> %s </b></p>
<p>Your home phone is: <b> %s </b></p>
<p>Your e-mail address is: <b> %s </b></p>
<p>You have opted for the <b>%s</b> course</p>
</BODY></HTML>'''
fs = cgi.FieldStorage()
title = fs['studtitle'].value
name = fs['studname'].value
dob=fs['studdob'].value
add=fs['studadd'].value
phone=fs['studphone'].value
email=fs['emailadd'].value
course=fs['studcourse'].value
print dynhtml % (title,name,dob,add,phone,email,course)
```

The preceding code accepts the data entered by the user in the details.html page and stores it in fs, which is an instance of the fieldstorage class in the cgi module. The dynhtml variable contains the Python code embedded in the HTML code to create the dynamic HTML page. Notice the Content-type tag in the script, which sends a header describing the contents of the document. This tag is used by the client browser and does not appear in the generated page. The values that this tag can be assigned are text/html, image/gif, text/plain, and image/jpeg. In the end, the code generates a dynamic Web page to display the information entered by the user. Figure 10.11 shows the dynamically generated page containing the data supplied by the user.

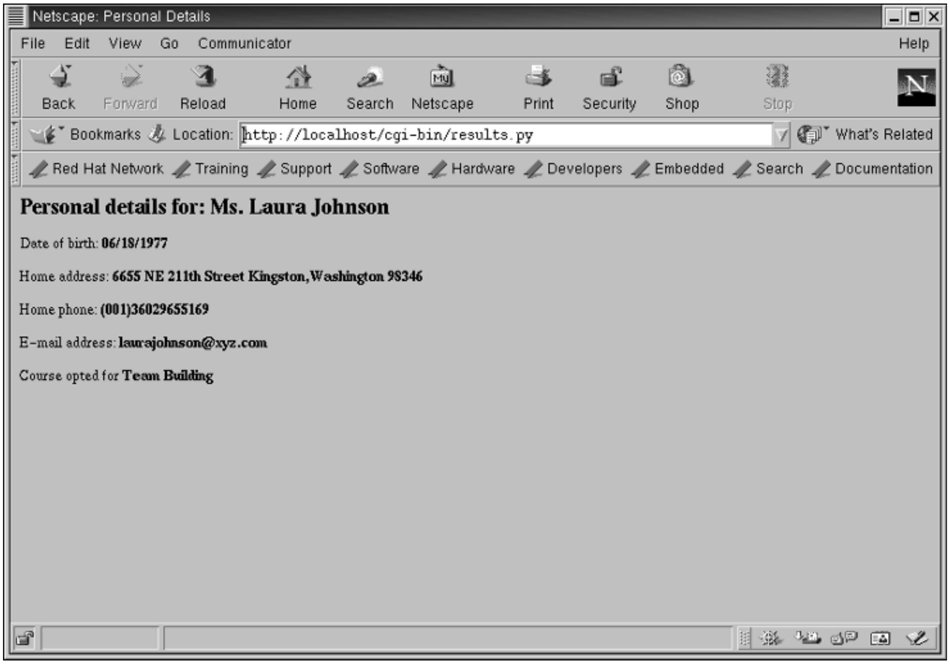


Figure 10.11 Results page in Netscape Navigator on Linux.

Result

Using what you have just learned, let's write the code for the `validate.py` script that checks whether a user has entered a login name and a password and displays a dynamically generated Web page based on the data entered by the user.

```
#!/usr/local/bin/python
import cgi
header= "Content-Type: text/html\n\n"
dynhtml='''<HTML><HEAD><TITLE>
%s </TITLE></HEAD>
<BODY><CENTER><HR><H2> %s </H2> <H3> %s </H3><HR></CENTER>
</BODY></HTML>'''
fs = cgi.FieldStorage()
passd="password"
if fs.has_key('login') and (fs['login'].value!=""):
     if fs.has_key('password'):
              fpass=fs['password'].value
```

```
                if fpass==passd:
                     abc="Connected"
                     message="Welcome...\n"
                else:
                     abc="Not connected"
                     message="Wrong password"
          else:
               abc="Not connected"
               message="Password not entered for"
          print header+dynhtml % (abc,message,fs['login'].value)
     else:
          abc="not connected"
          message="You have not entered a login name."
          message2="Click Back"
          print header+dynhtml % (abc,message,message2)
```

Write the CGI Program to Generate Both the Form and Results Pages

Let's now combine the HTML code in details.html and the CGI script in `results.py` into a CGI script `formresults.py`. This script will now display the form to accept the user input and display the results page. This means that both the pages will be generated dynamically. The following code represents `formresults.py`.

```
#!/usr/local/bin/python
import cgi
header= "Content-Type: text/html\n\n"
formhtml='''<HTML>
<HEAD>
<TITLE>Login Page</TITLE>
</HEAD>
<BODY>
<HR><CENTER>
<FORM method="POST" action="http://localhost/cgi-bin/formresults1.py">
<p>Login Name:<input type="text" name="login" value=""></p>
<p>Password: <input type="password" name="password" value=""></p>
<p><input type="submit" value="Submit">
<input type="reset" value="Reset"></p>
</FORM>
</CENTER>
<HR>
</BODY>
</HTML>'''
```

```
def show_form():
     print header+formhtml
dynhtml='''<HTML><HEAD><TITLE>
%s </TITLE></HEAD>
<BODY><CENTER><HR><H2> %s </H2> <H3> %s </H3><HR></CENTER>
</BODY></HTML>'''
fs = cgi.FieldStorage()
passd="password"
if not fs:
     show_form()
elif fs.has_key('login') and (fs['login'].value!=""):
     if fs.has_key('password'):
               Ch=0
               fpass=fs['password'].value
               if fpass==passd:
                    abc="Connected"
                    message="Welcome...\n"
               else:
                    abc="Not connected"
                    message="Wrong password"
     else:
          abc="Not connected"
          message="Password not entered for"
     print header+dynhtml % (abc,message,fs['login'].value)
else:
     pass
```

Execute the Code

To execute the **formresults.py** script, perform the following steps:

1. Save **formresults.py** in the `/var/www/cgi-bin` directory.
2. Type the following command:

   ```
   $ chmod +x /var/www/cgi-bin/formresults.py
   ```
3. In the address bar of Netscape Navigator, enter the following URL:

   ```
   http://localhost/cgi-bin/formresults.py
   ```
4. On the login screen, enter the login name `User1` and password `password`.
5. Click the Submit button. A results page is generated as shown in Figure 10.12.
6. Go back to the login page.
7. Enter the login name `User1` and click the Submit button. An error page is generated as shown in Figure 10.13.



Figure 10.12 The results page when the correct details are entered on the login page.



Figure 10.13 The results page when the password field is left blank.



Figure 10.14 The results page when an incorrect value for the password is entered.

8. Go back to the login page.
9. Enter the login name `User1` and the password `pass`. An error page is generated as shown in Figure 10.14.

Summary

In this chapter, you learned the following:

- The Internet is a connection of many other networks across the globe. It uses the TCP/IP protocol to transfer data across the networks.
- *World Wide Web (WWW)* is a common set of protocols that provides standards for specific computers to distribute documents on the Internet.
- Web browsers are programs that communicate with the Web servers on the Internet, enabling the download and display of requested Web pages.
- The address bar is a browser-specific element that is used to specify the URL of the Web page. The URL contains the name and address of the requested Web page.

- A typical HTTP transaction between a Web browser and a Web server will take place in the following manner:
  1. A TCP/IP connection is established between the client (browser) and the server.
  2. The browser sends a request for a particular HTML page.
  3. The server locates the file and sends a response in the form of the text content of the requested page.
  4. The TCP/IP connection is closed.
- The HTTP request is sent to the server along with the URL of the requested page, which is typed in the address location bar of the browser. The `GET` and `POST` methods are commonly used to specify the type of requests of HTTP 1.1.
- You can use HTML along with Python to create attractive and dynamic Web pages. HTML tags are used to contain specific elements of HTML.
- The `INPUT` element is specified within the `FORM` element in an HTML form. The main purpose of using the `INPUT` element is to accept user-specific input. The `INPUT` element helps developers create text boxes, buttons, check boxes, and radio buttons in their forms.
- Server-side scripting is used when there is a need to develop active Web sites that can interact with databases and allow the customization of the content of a Web page for each user.
- The mechanism of the server receiving the form, contacting the external program to process the request, and receiving and returning the newly generated HTML file is called Common Gateway Interface (CGI), and the external program that processes the client request is called a CGI script.
- The `cgi` module has to be imported in any CGI script written in Python. The `fieldstorage` class in the `cgi` module is responsible for communication with a client.
- The CGI application that processes the client request should be able to send the results back to the server in the form of a Web page. For this purpose, the CGI application should contain the code to generate an HTML page dynamically. This code is made up of the HTML code with the Python code embedded in it.
- CGI can be used to create both the form that accepts the user details and the dynamically generated HTML page to display the results after processing the information in the form.

CHAPTER

11

Database Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Define a database**
- **Identify the significance of databases**
- **Explain database-related concepts**
- **Identify the Python DB API**
- **Install MySQL**
- **Identify the features of MySQL**
- **Use MySQL in Python**

Getting Started

Until now, this book has familiarized you with the Python concepts that are important for developers or programmers to grasp before they can start using Python for Web development. Chapter 10, "CGI Programming," explained the basics of CGI. In the last chapter, you learned how to accept data from a user in an HTML form, process it, and

display the result back in a dynamically created Web page. You will agree that most transactions on the Web require the use of stored data sources that can be accessed and manipulated to yield desired output. Consider a simple example of a site that provides free e-mail services to its users. Have you ever wondered where the information specific to each of these users is stored? Or, for that matter, how the password specific to a user is validated every time the user tries to log on to the site? The answers to all these questions are databases. Databases give developers the ability to create well-formatted and structured data repositories. They enable data accessibility and availability across networks.

This chapter assumes that the reader has a basic knowledge of databases and understands how data is stored in databases. It also assumes that you are familiar with RDBMS concepts and their implementation in MySQL. For those of you who are new to MySQL, this chapter details concepts about installing MySQL and working with the databases and tables in MySQL.

This chapter also discusses the Python Database API. Next, the chapter explains the processes of accessing and manipulating a MySQL database by using Python commands. Finally, the chapter will discuss concepts such as the creation of a database table to store information and the use of query statements to access and manipulate data in the Techsity University scenario.

Before discussing database programming in Python, let's recap database management concepts.

Database Management

A database can be defined as a repository that stores related information. Examples of databases are formal databases for quantitative analysis, such as databases containing census data, or informal databases, such as those containing recipes, shopping lists, or task lists. Desired information can be extracted from voluminous data by using queries. Queries provide a quick, interactive way to retrieve information from a database.

Relational Database Management System

As discussed earlier, databases provide a methodology to structure and organize large amounts of data. This data can consist of details about a shopping cart, an online bank, a picture gallery, or a corporate network. A database management system (DBMS) presents a software mechanism to access, retrieve, and manage the data in a database in the form of tables consisting of rows and columns.

When a DBMS can retrieve information by using the data in the specified columns of a table to find additional data in another table, it is known as a relational database management system (RDBMS). A *relational database management system* allows you to define relationships between the tables present in a database. This improves speed and flexibility, and data from different tables can be combined in response to a request from a client. MS-Access, MS-SQL Server, Oracle, Sybase, Informix, and Ingress are some examples of DBMSs and RDBMSs that are available today. The access and retrieval of data from a database is achieved by the use of Structured Query Language, or SQL, in the standardized query format decided by International Organization for Standardization (ISO) and American National Standards Institute (ANSI).

Python Database API

The idea of providing a standard way in which different Python modules can access databases led to the development of Python Database API. In this way, consistency can be achieved among modules. Therefore, modules are easily understood, code is portable across databases, and database connectivity from Python is easy. The latest version of the Python Database API is 2.0. The specification of the Python Database API interface consists of several sections:

- Module Interface
- Connection Objects
- Cursor Objects
- Type Objects and Constructors
- Implementation Hints
- Major Changes from 1.0 to 2.0

For more information on Python Database API you can refer to: www.python.org/topics/database/DatabaseAPI-2.0.html.

Python Database API is maintained by the Database Special Interest Group (DB-SIG). For more information on SIGs, check out the site, www.python.org/sigs/db-sig/. Python Database API supports a wide range of database servers:

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

NOTE Although Python supports a wide range of database servers, because of its advantages, we will use MySQL for all database-related activities in this book.

Introduction to MySQL

MySQL is a powerful RDBMS developed by a commercial company called MySQL AB. The features of MySQL are as follows:

- In MySQL, you can have multiple related tables in a database, which makes it a powerful RDBMS.

- It is freely downloadable from its official site, www.mysql.com. Its source code can be used to customize it based on the need of the user.
- It has built-in support for a lot of applications and languages, such as Python, Perl, and PHP.
- It can be installed on almost any operating system.
- It consists of a thread-based memory allocation system, which is fast.
- It supports fixed-length and variable-length records.
- It has an efficient security system that allows host-based verification. Passwords are encrypted during transit, ensuring maximum security.
- It is capable of handling very large databases.
- Support is available on almost all MySQL programs. On Linux systems, the –`help` argument can be used with the program name to display a page that consists of online help. On Windows, MySQL documentation is installed with MySQL by default.
- Column types, such as `FLOAT`, `CHAR`, `TEXT`, `DATE`, `TIME`, `DATETIME`, `VARCHAR`, `YEAR`, `TIMESTAMP`, and others, are supported in MySQL.
- There are no reported memory leakages in MySQL.
- It consists of an optimized class library. All MySQL functions are implemented using this class library and, as a result, are very fast.
- It supports TCP/IP sockets, Unix sockets, and Named Pipes for connectivity. These concepts are discussed in Chapter 12, "Network Programming."

Installing and Configuring MySQL

If you have chosen Linux as your platform for MySQL and installed Linux with the Custom-Everything option, you do not need to install MySQL separately; MySQL comes bundled with Red Hat Linux. There are a few steps that you need to perform before you can start using MySQL. The steps are as follows:

1. Ensure that you have logged in as a root user.
2. On the command prompt, type the command `ntsysv`. The Services screen appears on the command prompt.
3. Now, select the option `mysqld` as shown in Figure 11.1, and click OK.
4. Type the following command at the command prompt to start the MySQL daemon:

```
# /etc/rc.d/init.d/mysqld start
```

Installing MySQL Separately

If not installed with Linux, MySQL can also be installed separately. It is recommended that you install MySQL by using its RPM files. The RPM files can be downloaded from



Figure 11.1 Services screen.

www.sourceforge.net. The RPM files will also be available on the Red Hat Linux 7.1 CD. Depending on your requirement, the necessary RPM files that you may want to use to install MySQL are the following:

`MySQL-<versionname>.i386.rpm.` This RPM file installs MySQL Server. You need to install this file if MySQL Server is not installed on a remote machine.

`MySQL-client-<versionname>.i386.rpm.` This RPM file will install the client-only version of MySQL. It is advisable that you always install this package along with MySQL Server.

`MySQL-devel-<versionname>.i386.rpm.` This RPM file will install all the necessary libraries and will include the files that are required to compile other client programs.

`MySQL-<versionname>.src.rpm.` This RPM file contains the entire source code for all the application packages mentioned here.

To install MySQL by using the RPM files, you need to use a single command as shown:

1. Download the latest RPM files or copy the required RPM files from the Red Hat Linux 7.1 CD to the `/root` directory.
2. Type the following command to install MySQL Server and a MySQL client:

```
# rpm -ivh MySQL-versionname.i386.rpm MySQL-client-
versionname.i386.rpm
```

After you install MySQL by using its RPM file, all necessary data is transferred to the `/var/lib/mysql` directory. Appropriate entries are also made in the `/etc/rc.d` directory. This configures the system to start MySQL at startup.

Working with MySQL

You have now learned to install and configure MySQL. Let's move on to learning the basics of MySQL so that you can start using MySQL. This chapter will briefly introduce two commands, `mysql` and `mysqladmin`. Primarily, these two commands are used to work with the databases in MySQL.

The `mysqladmin` Command

The `mysqladmin` command is used in MySQL for server administration. There are many things a database administrator can do using the `mysqladmin` command at the command prompt. The syntax of the msqladmin command is this:

```
mysqladmin [options] command1 command2......commandn
```

Some of the options that you can use with the `mysqladmin` command are listed in Table 11.1.

Table 11.2 lists the commands used with the `mysqladmin` command.

Table 11.1 Options Used with `mysqladmin`

OPTION	DESCRIPTION
`-?, —help`	This option is used to display help and exit.
-h, `—host=[Hostname]`	This option is used to connect to the host `Hostname`.
`-p[password], —password[=password]`	This option is used to specify the password when connecting to the server. If password is not specified, MySQL Server asks for a password from the tty.
`-P, —port=[portno]`	This option is used to specify the port number `portno` for connection.
`-s, —silent`	This option is used to exit silently if a connection to the server cannot be established.
`-u, —user=[user]`	This option is used to specify the user for login if not the current user.

Table 11.2 Commands Used with mysqladmin

COMMAND	DESCRIPTION
Create <database_name>	This command creates a new database with the specified name.
Drop <database_name>	This command deletes the database with the specified name.
Extended-status	This command displays the status in a table with two columns, Variable_name and Value.
Flush-hosts	This command clears all hosts that are cached.
Flush-logs	This command clears all server logs.
Flush-tables	This command clears all tables.
Flush-threads	This command clears all threads from the thread cache.
Ping	This command checks whether the MySQL daemon is functional. A message mysqld is alive is displayed if mysql is functional.
Refresh	This command clears all tables and then opens and closes log files.
Status	This command displays a short status message on the command prompt.
Variables	This command displays all available variables on the command prompt.
Version	This command displays the version information of the MySQL Server.
Shutdown	This command shuts down the MySQL server.

Creating a Database

To create a database, use the create command at the command prompt. For example,

```
#mysqladmin create Student
```

The preceding command creates a database with the name Student.

The `mysql` Command

The mysql monitor is the command-line interface used to interact with MySQL databases. The mysql monitor can be invoked at the command prompt as follows:

```
# mysql [OPTIONS] [database]
```

Without any option, the mysql monitor can be invoked at the command prompt as follows:

```
# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.23.36
Type 'help;' or '\h' for help. Type '\c' to clear the buffer
mysql>
```

You can specify the database to be used by using the database parameter. Many options can be specified with the mysql command. Some of the options that can be specified are listed in Table 11.3.

You can now use the MySQL interface to work with databases; you can create, modify, and delete tables and query and retrieve data from tables. However, databases can be created only on a machine that is running MySQL Server by using the mysqladmin command.

Table 11.3 Options Used with mysql

OPTION	DESCRIPTION
-?, —help	This option is used to display help and exit.
-B, —batch	This option is used to print results with a tab as a separator, each row on a new line. This option does not use the history file.
-D, —database=[databasename]	This option is used to specify the database to use.
-h, —host=[Hostname]	This option is used to connect to the host Hostname.
-p[password], —password[=password]	This option is used to specify the password when connecting to the server. If the password is not specified, MySQL Server asks for a password from the tty.
-P, —port=[portno]	This option is used to specify the port number portno for a connection.

Table 11.4 The Basic MySQL Commands

COMMAND	WHAT IT DOES
`Use`	Makes the specified database the current database.
`Create table`	Creates a table in a database.
`Select`	Retrieves records from a table.
`Insert into`	Inserts new records in a table.
`Update`	Modifies the values in the records of a table.
`Delete`	Deletes records from a table.
`Drop table`	Deletes a table from a database.

Let's briefly look at some basic MySQL commands and their utility. Table 11.4 lists each of these commands to demonstrate their implementations.

Let's now look at the implementations of the SQL commands by using examples.

Specifying the Database to Be Used

If you have not specified the database to use while invoking the `mysql` monitor, you can use the `use` command. For example,

```
mysql> Use Student;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

Creating a Table

The `create table` command is used to create a new table in a database. The following syntax illustrates the line of code used for creating a table:

```
create table tableName (columnName1 datatype, columnName2 datatype...);
```

Consider a situation in which you need to create a table in the `Student` database, which is used to validate the authenticity of the username and password entered by a student on the login page. The fields for this table are `cStudent_Id` and `cPassword`.

The following command shows the use of the `create table` command to create the `Login` table in the `Student` database:

```
mysql> create table Login (cStudent_Id char(10), cPassword char(10));
```

Notice that the data type used in the preceding example for the two fields is `char`. Table 11.5 lists the data types available in MySQL.

Table 11.5 Data Types Available in MySQL

DATA TYPE	DESCRIPTION
Numeric types	
`NUMERIC`	This data type represents a number.
`DECIMAL (precision,scale)`	This data type represents a decimal number. `Precision` specifies the number of significant decimal digits that will be stored for values, and `scale` specifies the number of digits that will be stored following the decimal point.
`INT`	This data type represents an integer from -2147483648 through 2147483647.
`SMALLINT`	This data type represents an integer from -32768 through 32767.
`FLOAT`	This data type represents a number from -1.79E+308 to 1.79E+308.
`REAL`	This data type represents a number from -3.40E+38 through 3.40E+38.
`DOUBLE PRECISION`	This data type represents a 64-bit value from 10^{308} through 10^{-323}.
Date and Time types	
`DATETIME`	This data type represents a date of the format `yyyy-mm-dd hh:mm:ss`.
`DATE`	This data type represents a date of the format `yyyy-mm-dd`.
`TIMESTAMP`	This data type represents dates ranging from the beginning of 1970 to sometime in the year 2037 with a resolution of one second. Values are displayed as numbers, and their lengths depend on their display sizes.
`TIME`	This data type represents time in the format `hh:mm:ss`.
`YEAR`	This data type represents a year in the format `yyyy`. The range is 1901 through 2155.
String types	
`CHAR(no_of_bytes)`	This data type represents strings of length 0 through 255. Regardless of the length of the string, a `char` type occupies the specified number of bytes.

DATA TYPE	DESCRIPTION
VARCHAR(no_of_bytes)	This data type represents variable-length strings of length 0 through 255. In contrast to CHAR, VARCHAR values are stored using only as many characters as are needed, plus one byte to record the length.
TEXT	This data type represents strings, which do not require a size to be specified and can be as long as you want.

To test whether the table was successfully created, type the following command:

```
mysql> show tables;
```

This command will display the tables present in the database named Student. The output of the command appears as shown:

```
mysql> show tables;
+------------------+
| Tables_in_Student|
+------------------+
| Login            |
+------------------+
1 rows in set (0.01 sec)
```

The preceding output indicates that a table named Login is created in Student. You can also view the fields that you had added to the table Login. To do this, type the following command:

```
mysql> explain Login;
```

This command helps an administrator keep track of the number of fields and their types. The output of the command appears as shown:

```
+-------------+------------+------+-----+---------+-------+
| Field       | Type       | Null | Key | Default | Extra |
+-------------+------------+------+-----+---------+-------+
| cStudent_Id | varchar(10) | YES |     | NULL    |       |
| cPassword   | varchar(20) | YES |     | NULL    |       |
+-------------+------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

Notice all the information about all the fields present in the table. In the case of the Student database we had created a table, Login, with two fields. All the fields in the table are displayed along with a description of each field.

Retrieving Records from a Table

The `select` command is used to view the records of a table. The following syntax illustrates the line of code for viewing the records of a table:

```
select * from tableName;
```

The use of an asterisk specifies the retrieval of all records from a particular table. You can also specify a selection criterion in the `select` statement. The `where` clause is used along with the selection criterion to specify a selective retrieval of records from a table. The syntax in such a case will be this:

```
select * from tableName where selectionCriterion;
```

The following statement shows the use of the `select` command to retrieve records for a student with the id `User15`:

```
select * from Registration where cStudent_Id="User15";
```

Inserting Records in a Table

The `insert into` command is used to insert new records in a table. The following syntax illustrates the line of code for adding records to a table:

```
insert into tableName values (columnValue1, columnValue2 ...);
```

It is important to maintain the sequence of the column values being inserted, which should be similar to the order of the column names specified during creation of the table. The following code snippet is used to insert the first name, last name, address, phone number, and e-mail address of a student in the `Registration` table:

```
insert into Registration values("Dunston", "Payne", "21, Sunley House,
 Eastern Avenue", "362283838383", "duns@xyz.com");
```

Modifying the Data in a Table

The `update` command is used to modify the records in a table. The following syntax illustrates the line of code for modifying a column based on a specified criterion:

```
update tableName set columnName="newColumnValue" where criterion;
```

The following code snippet is used to modify the last name of Betty from Smith to Charles in view of her recent marriage:

```
update Registration set lastName="Charles" where firstName="Betty"
```

The `alter` command is used to modify the columns of a table. The following tasks can be performed using the `alter` command:

Adding a column to a table. To add a column to a table, use the following syntax:

```
alter table table_name add newcolumn fielddefinition;
```

Changing the type of a column. To change the type of an existing column, use the following syntax:

```
alter table table_name change column_name newfielddefinition;
```

Indexing a column in a table. To index an existing column, use the following syntax:

```
alter table table_name add index column_name (column_name);
```

Making a unique column in a table. To make a unique column in a table, use the following syntax:

```
alter table table_name add unique column_name (column_name);
```

Deleting a column from a table. To permanently delete a column from an existing table, use the following syntax:

```
alter table table_name drop column_name;
```

Deleting Records from a Table

The `delete` command is used to delete records from a table. The following syntax illustrates the SQL statement used for deleting all the records of a table:

```
delete from tableName;
```

The `where` clause is used along with a selection criterion to specify a selective deletion of records from a table. The syntax in such a case will be this:

```
delete from tableName where selectionCriterion;
```

The following code snippet shows the use of the `delete` command to retrieve records for a student named Jonathan:

```
delete from Registration where firstName="Jonathan";
```

Deleting a Table

The `drop table` command is used to delete a table from a database. The following syntax illustrates the SQL statement used for deleting a table:

```
drop table tableName;
```

The following code snippet shows the use of the `drop table` command to delete the `tempTransaction` table:

```
drop table tempTransaction;
```

The concepts covered up to now discussed the basics of MySQL. Let's move on to understanding how the preceding concepts can be used to manipulate the data in a database.

Accessing a Database from a Python Script

Problem Statement

The online site of Techsity University has a provision for new students to register online. All that students need to do is register themselves by filling out the online registration form. The administrator of the Web site processes the request, and if all the essential criteria for having an account with the University are met, the student is duly informed and the database is updated with the new student's details.

Your colleague John has already designed the interface for the page that will be used to update the database with a new student's details. You, as a developer, have been assigned the task of writing the code for the database interaction. The code, when executed, will insert a new student's details into the related database table. The prototype for this particular application that inserts new records into the registration first requires the creation of a registration table before the insertion of records.

NOTE The format and appearance of the HTML forms used for accepting students' details will change as the application progresses toward completion.

The following HTML code for the registration page accepts input for the student registrations:

```
<HTML><HEAD><TITLE>
User Registration Form
</TITLE></HEAD>
<BODY>
<b><font size="5"><u>User Registration Form</u></font></b>
<FORM method="POST" ACTION="http://localhost/cgi-bin/regdetcgi.py">
  <p>Name:
    <INPUT TYPE=text NAME=studname VALUE="" SIZE=30>
  <p>Date of Birth:
    <input type="text" name="studdob">
    (yyyy-mm-dd) </p>
  <p>Address:
    <textarea NAME=studadd rows=2 cols=30></textarea>
```

```
    </p>
    <p>Country:
      <input type="radio" name="studcountry" value="U.S.A" checked>
      U.S.A
      <input type="radio" name="studcountry" value="Canada">
      Canada
      <input type="radio" name="studcountry" value="Other">
      Other
    </p>
    <p>Home phone #:
      <INPUT TYPE=text NAME=studphone VALUE="" SIZE=30>
    </p>
  <p>E-mail address:
  <INPUT TYPE=text NAME=emailadd VALUE="" SIZE=30></p>
    <P>
      <INPUT TYPE=submit>
      <INPUT TYPE=RESET>
    </P>
  </FORM>
  </BODY></HTML>
```

Figure 11.2 displays the registration page.

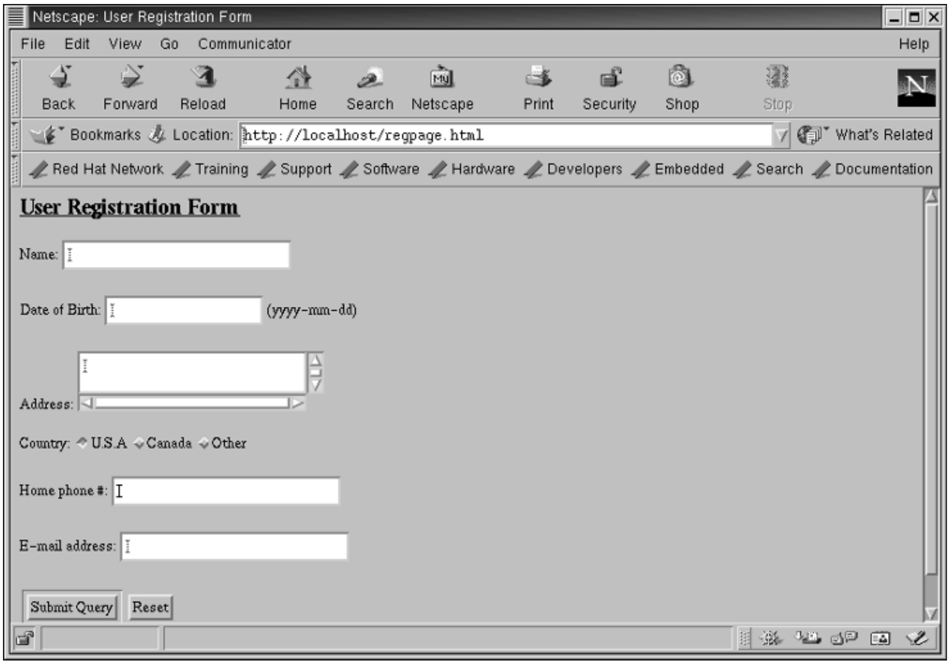


Figure 11.2 The registration page to enter user details.

Task List

- ✓ **Identify the elements of the table that stores registration details.**
- ✓ **Identify the steps for connecting to the database.**
- ✓ **Write the code to create a table in the database.**
- ✓ **Write the code to insert the registration details to the table created.**
- ✓ **Execute the code to create the table in the database.**
- ✓ **Execute the code to insert data into the table.**
- ✓ **Verify the data in the database.**

Identify the Elements of the Table That Stores Registration Details

Based on the elements in the input page, we will create the registration table to accept the student details. Table 11.6 represents the composition of the `regdetails` table.

Identify the Steps for Connecting to the Database

Before you can begin manipulating the data held in the tables of the database, you first need to execute the following phases:

Import the `MySQLdb` module. This phase makes the `MySQLdb` module available to the Python application.

Connect to the database. This phase establishes a connection with the database. This involves two steps: first, creating a connection object and then creating a cursor object to execute an SQL statement.

Query the database. This phase involves executing the SQL statement. If the SQL statement returns results—for example, the results of a select statement—this phase also involves retrieving the results.

Table 11.6 Elements of the Registration Table

TABLE FIELD NAME	CONTENTS	DATA TYPE
`cName`	Name of the student	`varchar(20)`
`cDob`	Date of birth of the student	`Date`
`cAdd`	Address of the student	`varchar(50)`
`cCountry`	Country where the student resides	`varchar(20)`
`cPhone`	Home phone number	`varchar(15)`
`cEmail`	E-mail id	`varchar(20)`

Import the `MySQLdb` Module

To write a program that can connect to a database, you first need to install and configure the `MySQLdb` module. The RPM file for this module can be obtained from http://dustman.net/andy/python/ MySQLdb. Each version of Python has a specific `MySQLdb` RPM file. After obtaining the RPM file, type the following command at the command prompt:

```
rpm -ivh MySQL-python-0.9.1-1py2.i386.rpm
```

You can import the `MySQLdb` module as follows:

```
>>>import MySQLdb
```

Connect to the Database

After importing the `MySQLdb` module, you need to identify the database within the module that will be queried and then connect to that database. To do this, you need to create a connection object by invoking the `connect()` method of the `MySQLdb` module. The following line of code illustrates the parameters used to establish a connection with a database that resides on the same machine as the code:

```
>>>Connection = MySQLdb.connect(host="localhost",db="Student",\
                 port=8000, username="laura", passwd="password")
```

After establishing a connection, you need to create a cursor object, which can be done using the `cursor()` method of the connection object.

The `cursor()` Method

This method returns a new cursor object and represents a database cursor. Cursors are used to manage all operations in a database. Operations in a database act on a complete set of rows. For example, the set of rows returned by a `select` statement consists of all the rows that satisfy the conditions in the `where` clause of the statement. This complete block of rows returned by the statement is known as the *result set*. Applications such as those in Python cannot always work effectively with the entire result set as a unit. Most applications need a mechanism to work with one row or a small block of rows at a time. Cursors provide such a mechanism to work with result sets. Cursors aid in positioning at specific rows in a result set and retrieving a row or a group of rows from the current position in the result set.

Therefore, a cursor object functions as the active connection to the database. A cursor object can be created using the previously created connection object as follows:

```
>>> con=connection.cursor()
```

After you are connected to a database, you can submit and retrieve the results of a query though the cursor object.

NOTE According to the Python Database API, a connection object should also support the following methods:

- `commit()`. **This method commits any pending transaction with a database. This feature is initially off for MySQL because MySQL supports an autocommit feature. This means that whenever an update to a table in a database is made, it is immediately executed and the changes are updated on the disk. If you want a set of statements to be executed together, the autocommit feature can be turned off using the following command:**

  ```
  mysql> SET AUTOCOMMIT=0
  ```

  After this, you must use `COMMIT` **to store your changes to disk or** `ROLLBACK` **if you want to ignore the changes you have made since the beginning of your transaction.**

- `rollback()`. **This method causes a database to roll back a pending transaction and start the transaction again. If you close the connection with the server without committing the transaction, the transaction is automatically rolled back.**
- `close()`. **This method closes a connection to a database. Any further command to access the database will return an error after the connection is closed.**

Query the Database

After a connection between the Python application and the database with which the application wants to interact is set up, you use methods and attributes of a cursor object to send simple queries to the database. Out of the methods available for a cursor object the `executeXXX()` methods are used to execute any MySQL statement, and the `fetchXXX()` methods are used to get the results of the previous `executeXXX()` as a list of tuples. These methods are discussed in detail next.

Executing MySQL Commands

The `execute()` method of a cursor object prepares and executes a database command or a query. The syntax of the `execute()` method is this:

```
execute(command[parameters])
```

For example,

```
>>>con.execute('insert into Login values("user1","pass123")')
```

You can also pass parameters to the `execute()` method if instead of actual values you want to use variables in the MySQL command. For example,

```
>>>stud_id="user2"
>>>password="password272"
>>> con.execute('insert into Login values("%s","%s")' % \
                 (stud_id,password))
>>> con.execute(select * from login)
```

In the preceding command, the values contained in the variables `stud_id` and `password` will be passed to the insert command. Therefore, instead of supplying the actual values, you can supply the variables to the execute command.

The `executemany()` command can also be used if you want to execute multiple operations by using a single command. In such a situation, you can write multiple query statements followed by the parameters in the `executemany()` command.

Retrieving Query Results

The following methods can be used to retrieve any results returned by the `execute-XXX()` method. `fetchXXX()` methods are discussed in detail as follows.

`fetchone()`. This method fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table. Consider the following call to the `execute()` method.

```
>>> con.execute(select * from login)
```

You know that this preceding command returns all the records in the `Login` table. All the records returned will be contained in a result set. Now, the `fetchone()` method can be used to fetch a single row of the result set as follows:

```
>>> result=con.fetchone()
>>> result
('user1', 'pass123')
>>> result=con.fetchone()
>>> result
('user2', 'password272')
```

Notice that when the `fetchone()` method is used the second time, it retrieves the second row of the result set. This means that the `fetchone()` method can be used to extract the next row in a result set.

`fetchall()`. This method fetches all the rows in a result set. If some rows have already been extracted from the result set, the `fetchall()` method retrieves the remaining rows from the result set. The following statement illustrates the use of the `fetchall()` method to retrieve all the records from the `Login` table.

```
>>> result=con.fetchall()
>>> result
(('user1', 'pass123'),('user2', 'password272'))
```

`rowcount`. This is a read-only attribute and returns the number of rows that were affected by an `executeXXX()` method.

Write the Code to Create a Table in the Database

The statements to create the `regdetails` table in the `Registration` database are as follows:

```
#!/usr/local/bin/python
import MySQLdb
connection=MySQLdb.connect(host="localhost",db="Registration",\
            user="root",passwd="new-password")
```

```
con=connection.cursor()
sql_stmt='create table regdetails (cName varchar(20),\
      cDob date, cAdd varchar(50), cCountry varchar(20),
      cPhone varchar(15),cemail varchar(20))'
try :
    con.execute(sql_stmt)
    print 'Table created'
except:
    print 'Cannot create table'
con.close()
```

Write the Code to Insert the Registration Details into the Table Created

Because the data that is to be added to the `regdetails` table is entered in an HTML form, a CGI script needs to be written to enable this addition.

```
#!/usr/local/bin/python
import cgi
import MySQLdb
print "Content-Type: text/html\n"
dynhtml='''<HTML><HEAD><TITLE>
Personal Details</TITLE></HEAD>
<BODY><HR><H2><center>Personal details for %s</H2>
<p>%s</p>
<HR>
</BODY></HTML>'''
fs = cgi.FieldStorage()
name = fs['studname'].value
dob=fs['studdob'].value
add=fs['studadd'].value
country=fs['studcountry'].value
phone=fs['studphone'].value
email=fs['emailadd'].value
try:
     connection=MySQLdb.connect(host="localhost",db="Registration",\
       user="root",passwd="new-password")
     con=connection.cursor()
     sql_stmt='insert into regdetails values\
("%s","%s","%s","%s","%s","%s")' % (name,dob,add,country,phone,email)
     con.execute(sql_stmt)
     message="Successfully entered in the database"
except:
     message="Error writing data to the table"
#result_set=con.fetchall()
con.close()
print dynhtml % (name,message)
```

Execute the Code to Create the Table in the Database

To be able to implement or view the output of the code to create the `regdetails` table, you need to execute the following predefined steps:

1. Save the file used to create a table as **createtable.py**.
2. At the shell prompt, type `python` followed by the name of the file if the file is in the current directory. Figure 11.3 displays the screen that represents the successful creation of the table in the database.

Execute the Code to Insert Data into the Table

To add student details to the `regdetails` table, perform the following steps:

1. Save the script used to insert registration details in the /var/www/cgi-bin directory as **regdetcgi.py.**
2. Type the following command:

   ```
   $ chmod +x /var/www/cgi-bin/regdetcgi.py
   ```

3. In the address bar of Netscape Navigator, enter the following URL:

   ```
   http://localhost/regpage.html
   ```

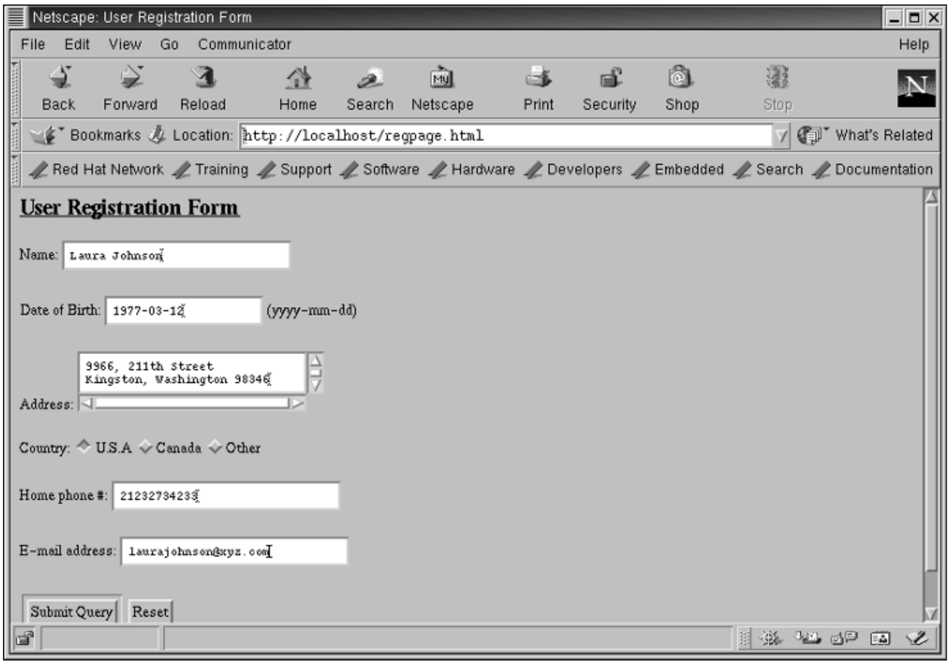


Figure 11.3 The screen message after creation of the table.

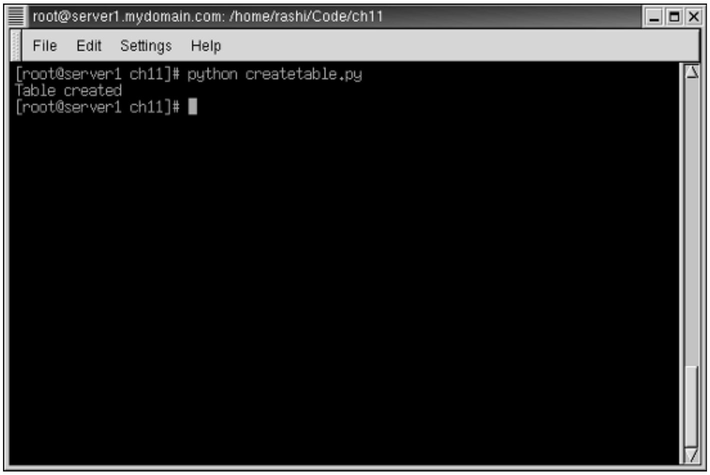


Figure 11.4 The registration page with input details.

4. On the registration screen, enter student details as shown in Figure 11.4.
5. Click the Submit button. The results page is generated as shown in Figure 11.5.

   If the query is not successfully executed, the page shown in Figure 11.6 is displayed.

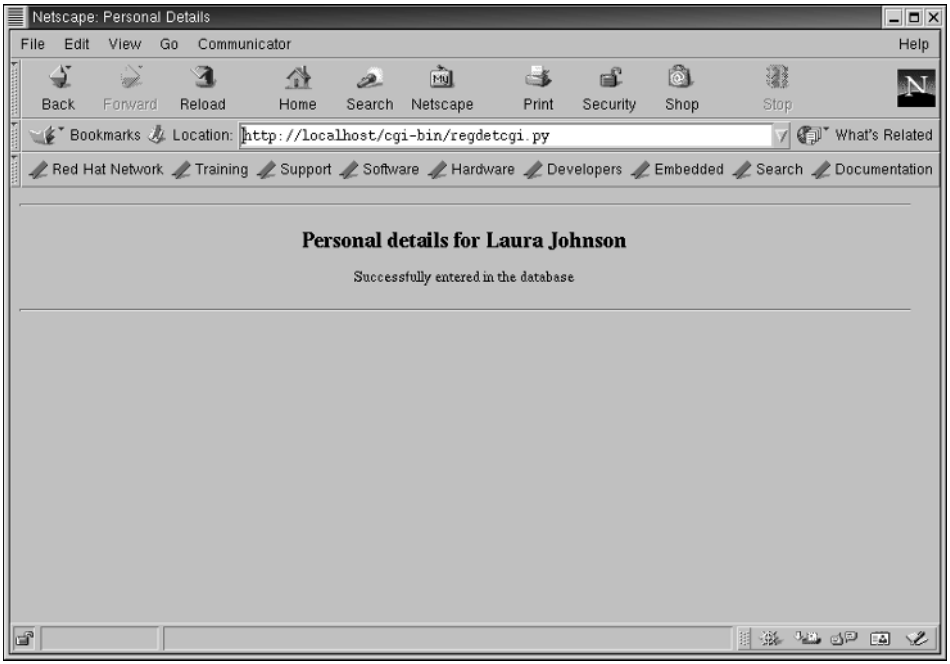


Figure 11.5 The results page when data is successfully inserted to the database.

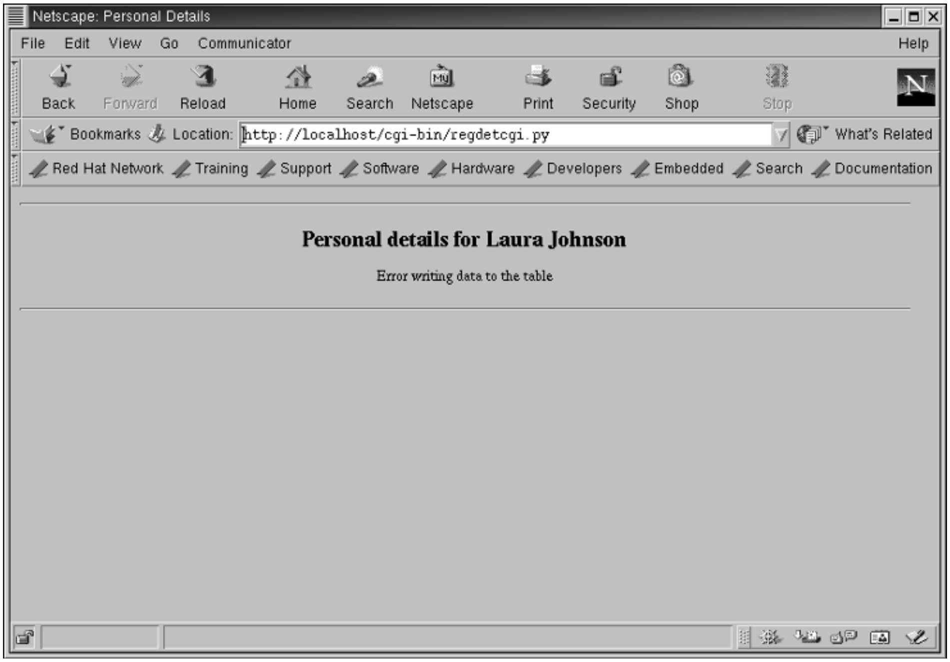


Figure 11.6 The results page when data is not inserted to the database.

Verify the Data in the Database

To verify that student details have been entered in the `regdetails` table, execute the following code:

```
#!/usr/local/bin/python
import MySQLdb
connection=MySQLdb.connect(host="localhost",db="Registration",\
         user="root",passwd="new-password")
con=connection.cursor()
sql_stmt2="select * from regdetails"
try :
    con.execute(sql_stmt2)
    print "Records in the table are: "
    count=con.rowcount
    print count
    i=0
    while i<count:
        result_set=con.fetchone()
        print "Name:", result_set[0]
        print "Date of birth:", result_set[1]
        print "Address:", result_set[2]
```

```
        print "Country:", result_set[3]
        print "Phone number:", result_set[4]
        print "Email id:", result_set[5]
        i=i+1
except:
    print 'Cannot display records'
con.close()
```

Summary

In this chapter, you learned the following:

- A database can be defined as a repository that stores related information. Databases provide a way to structure and organize large amounts of data.
- A database management system (DBMS) presents a software mechanism to access, retrieve, and manage the data in a database in the form of tables consisting of rows and columns. When a DBMS can retrieve information by using the data in a table to find additional data in another table, it is known as a relational database management system (RDBMS).
- The idea of providing a standard way in which different Python modules can access databases led to the development of Python Database API. Python Database API is maintained by Database Special Interest Group (DB-SIG).
- MySQL is a free, multithreaded RDBMS that can be used to interact with Python applications.
- MySQL comes bundled with Red Hat Linux; however, MySQL can also be installed separately. The RPM file for installing MySQL can be downloaded from www.sourceforge.net.
- The `mysqladmin` command is used in MySQL for server administration. To create a database, use the `create` command at the command prompt. For example,

  ```
  #mysqladmin create Student
  ```
- The `mysql` monitor is the command-line interface to interact with MySQL databases. The `mysql` monitor can be invoked at the command prompt as follows:

  ```
  # mysql [OPTIONS] [database]
  ```
- The various commands that can be used to access and modify tables in a MySQL database are: `use`, `create table`, `select`, `insert into`, `update`, `delete`, and `drop table`.
- MySQL allows the use of a host of data types, such as NUMERIC, DECIMAL, INT, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, DATETIME, DATE, TIMESTAMP, TIME, YEAR, CHAR, VARCHAR, and TEXT.

- Before you can begin manipulating the data held in the tables of the database, you first need to execute the following steps:
  1. Import the `MySQLdb` module
  2. Connect to the database
  3. Query the database
- To write a program that can connect to a database, you first need to install and configure the `MySQLdb` module. The RPM file for this module can be obtained from http://dustman.net/andy/python/ MySQLdb.
- To connect to a database, you need to create a connection object by invoking the `connect()` method of the `MySQLdb` module. For example,

```
>>>Connection = MySQLdb.connect(host="localhost",db="Student",\
port=8000, username="laura", passwd="password")
```

- The `cursor()` method returns a new cursor object and represents a database cursor. Cursors are used to manage retrieval operations in a database. A cursor object can be created using the previously created connection object as follows:

```
>>> con=connection.cursor()
```

- The methods that can be used to extract records from a result set are: `execute()`, `executemany()`, `fetchone()`, and `fetchall()`.

CHAPTER

12

Network Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Use the socket module to:**
  - **Create a TCP server and a TCP client**
  - **Execute a TCP server and a TCP client**
  - **Create a UDP server and a UDP client**
  - **Execute the UDP server and the UDP client**
- **Use other network programming-related modules**

Getting Started

In the previous chapters, you have learned about the various components and techniques of programming in Python. In this chapter, you will learn about network programming in Python by using sockets. We will start with a discussion on client/server architecture, followed by an overview of related concepts, such as protocols, sockets, IP addresses, and ports. This will give you the foundation to work with network programming modules of Python and use them to create network applications.

This book introduced client/server architecture in Chapter 10, "CGI Programming." Let's discuss client/server architecture in detail in the following section.

Client/Server Architecture

Networking is the technology that has changed the way information flows in present times. It has made communication possible between computers over a LAN, a WAN, and above all, the Internet. One of the important components of networking that has made it so successful is client/server architecture.

Client/server architecture is a term used to define the communication process between two computers, one of which is a client and the other is a server. What is a client? What is a server? Let's consider an example to explain this concept:

Consider that you are on vacation in the Caribbean. The weather is fine, and you are enjoying your holidays. One day, you learn that the hotel is organizing a charity event and that you have been invited to it. There is a dress code for the event is formal, and you are required to wear your best formal clothes. You are on a holiday, though, and do not have a tuxedo with you. You immediately call the concierge desk of the hotel and are greeted by a beautiful female voice. You ask the lady to get you one size 42, single-breasted black tuxedo with classic peak lapels, a white shirt, and black trousers with a double-pleated front. You also ask her to get the matching accessories. You tell her to send everything to your room by 5:00 in the evening. Around 5:00 P.M., you receive all the items you requested. You neither know, nor do you want to know, how all the items were procured.

The entities involved in this example are the tuxedo, the lady at the concierge desk who got the tuxedo, and, of course, you. You are the customer or client who ordered the tuxedo. There was a process involved in getting the tuxedo to you, which you initiated. Your request was processed, and the items were delivered to you.

This is how client/server architecture works. A client places a request or order to a server. The server processes the client's request. In client/server architecture, communication between a client and a server is most important. Both the client and the server can be on the same computer or on separate computers on the network. Usually, client/server communication takes place over the network.

A server provides services required by one or more clients. The server listens for the client requests, services the requests, and again waits for more requests. A successful transaction in client/server architecture begins when a client asks a server for a particular service and ends after the server replies to the client request. To allow such transactions to be repeatedly successful, the server should always be available to service client requests. Each client request is a separate transaction, which ends when a server replies successfully to the request or returns an appropriate reason for a failure.

Client/server architecture applies to both hardware and software systems. Therefore, the meaning of client/server architecture depends on the system you are describing and where you are applying this terminology. You can describe client/server architecture as application development architecture in which the client requests services and the server services these requests. Here, both the server and the client can be hardware components or programs running on the same or different computers.

File and print servers are common examples of servers in the hardware client/server architecture system. In the case of a print server, a client sends a print job, and a print

server processes the request and redirects it to a printing device. The print server and the client can be on the same or different computers. In client/server architecture, this does not affect the transaction between the client and the server. Client/server architecture is designed to separate the presentation of data from its internal processing and storage. The complete transaction of a client sending a request and getting a reply is hidden from a user.

In a software client/server architecture system, the client and the server are not hardware devices but programs running on some hardware. Software servers are commonly used for data storage, data retrieval, program execution, and working with data in various other ways. Web servers and database servers are common examples of servers in the software client/server architecture system.

In a client/server application, the server manages the shared resources that are accessed by multiple users. The Web server is an ideal example of a software server. It delivers HTML pages to different users across the Internet. These users can be on different platforms and may be using different types of software to access the Web server. The users are able to transact with the server without being aware of these complications because client/server architecture keeps its internal processing separate from data presentation.

Database servers are another example of a server in a software client/server architecture system. A client sends a request to a database server to obtain data or store data. An example of such a software client/server architecture system is the airline ticket reservation system. When you request a ticket for a particular flight on a specific date, this information is sent to a database as a query, and the database either returns a list of flights that match your requirements or displays an appropriate message.

Protocols

In all forms of communication, there are a few rules that have to be followed. This is true even for network communication between computers. The rules used for network communication are called network protocols. Protocols are a set of rules and standards that one computer needs to follow in order to communicate with another system over the network. In client/server architecture, the manner in which a client contacts a server and the way in which the server replies are defined by the protocol being used.

Different protocols can be used for network communication. These protocols can be categorized depending on the type of network connection they support. Protocols can be connection-oriented, such as TCP, or connectionless, such as UDP.

Python supports both types of protocols. Later in the chapter, you will use Python's network-programming components to enable network communication by using both types of protocols.

Network Programming

A server should always be ready to receive client requests. To do this, the server should have a communication point available for receiving client requests. This communication point should always be ready to receive client calls. For a client to contact this communication point, the client should be aware of it. It would be of no use to have a communication point that is not known to the clients. It would be like moving to a new

house and waiting for your mail messages to reach you there, without your telling anyone about the new address.

On the other hand, it is as important for a client to know the address of a server's communication point. When the client wants to communicate with the server, it will create a communication point, which will call the communication point of the server. After the server's communication point is found, the data is transmitted. After the transaction is over, the client disconnects from the server and closes its communication point. On the other hand, the server keeps the communication point open at its end, ready to receive more client requests.

Sockets

Sockets are software objects that allow programs to make connections. They are data structures that enable network communication by using protocols, such as TCP/IP and UDP. As described earlier in the chapter, a client and a server require a communication point to communicate between themselves. Sockets are the communication points that act as endpoints in two-way communication between the client and the server, where the client and the server are two programs running on the network. To establish a connection for network communication, a request is sent by a client to the server socket. After the connection is established, the transactions can take place. After the transaction is over, the client socket disconnects itself from the server.

Sockets were first introduced in 1981 as part of the BSD flavor of Unix. They provided an interface for communication between Unix systems over the network. They were originally used for Inter Process Communication (IPC) between two programs on the same host or platform. Because they were first introduced with Unix BSD 4.2, they have come a long way and have become very popular. Sockets are the only IPC forms that support cross-platform communication. They have become an important component in the development of the Internet as a platform-independent system, making it easy for computers around the world to communicate with each other.

Sockets are bound to ports, which are numeric addresses. A server receives client requests through ports. In order to connect to the server, a client should know the host name or IP address of the computer on which the server is running and the port number of the server. For example, you can compare the host name to the address of the building in which you live and the port to your apartment number in the building. A person who wishes to reach your apartment must know the address of your building and the apartment number.

A few port numbers are assigned to specific protocols. When you are working with these protocols, if you don't specify the port number, the default port number is used. A port is an entry point through which an application or a service residing on a server is accessed. It is a 16-bit integer, which can range between 65535. But, you can use a port number greater that 1024 freely inside your programs. This is because the range 0-1023 are reserved by the operating system for the network protocols. Table 12.1 shows the port numbers associated with common protocols.

Table 12.1 Protocols and Default Port Numbers

PROTOCOL	PORT NUMBER
FTP	21
Telnet	23
SMTP	25
BOOTP	67
HTTP	80
POP	110

Let's run through the complete process of network communication between a client and a server.

Before the actual process of communication can start, the server has to be made ready to receive client requests. It should be running on a specific computer and should have a socket that is bound to a specific port number.

When the server is ready and listening for the client request, the client can try to make a connection. To do this, the client tries to contact the server socket on the computer on which the server is running by using the host name or IP address of the computer and the server port number. The client should have this information about the server if it wants to connect to the server.

When a server receives a request for connection, the server may accept the request. When a server accepts a client request for connection, a new socket with a new port is assigned to the server, leaving the original port open. This allows the server to accept new client requests without affecting the existing connection. The new port is temporary and connection specific. As soon as the connection ends, the port is released.

The server can be compared to a switchboard operator at a company's corporate office. When the operator receives a call from someone, it is similar to the server receiving a client request for connection. The operator asks you about the person with whom you want to talk and transfers the call to that person. This is similar to a server accepting the connection and assigning it to a new port, leaving the original port open. After your call has been transferred, the operator is free to receive other calls on the same line. In a similar manner, after the server accepts the connection and is assigned a new port, the original port is ready to receive more requests from other clients.

When the server accepts the client's request for connection, a socket is created at the client end. This socket is assigned a local port number by the client computer. The client then communicates with the server by using this socket. Therefore, a socket is a communication channel between the client and the server that can be used by both of them to exchange data and perform other tasks.

Now that you know about the fundamental concepts of network communication and programming, let's learn to create network programs in Python.

Using Sockets

Problem Statement

Techsity University has an IP network. Its IT department has suggested implementing client/server architecture for university systems over the IP network. The management of the university is not very sure about converting all its systems to client/server architecture. They are also not sure about the integrity of exchanging data in client/server architecture. Therefore, they have planned to have a pilot implementation. The university has created a team, led by Jenny, to conduct the pilot.

The main concern of the team is to demonstrate the use of client/server architecture to management and to maintain the security of data sent over the network. The team has therefore decided to implement socket programming to show the working of client/server architecture. For the pilot, data will be exchanged between the computers in the Admissions office and the IT department. The computer in the IT department will be the main computer and will store data in a file.

Task List

- **Identify the type of sockets to be used.**
- **Write the code to run on the IT department computer.**
- **Write the code to be run on the Admissions office computer.**
- **Execute the code created for the IT department computer.**
- **Execute the code created for the Admissions office computer.**
- **Verify that data has been saved to a file in the IT department computer.**

Identify the Sockets to Be Used

As discussed earlier, before a server can be made ready to listen to client requests, it should have a socket bound to a specific port number. You also need to create a socket at the client end to allow the client to make a connection with the server. The following section discusses how you can create sockets by using Python.

The `socket` Module

To implement network programming in Python by using sockets, you use the `socket` module. It contains various methods used for socket-based network programming. The most common of them is the `socket()` method.

The `socket()` method is used to create a new socket. It returns the socket object, which is an instance of the `SocketType` class. The syntax of the `socket()` method is as follows:

```
socket(family, type, protocol)
```

You can have the `family` value as AF_UNIX or AF_INET. AF in AF_UNIX and AF_INET stand for Address Family. These family names define whether the client and server programs run on the same or different computers. The sockets of the AF_UNIX family are also called Unix sockets and were used originally in Unix BSD, the flavor of Unix that introduced sockets. You use the sockets of this family for interprocess communication on the same computer.

With the growth of networks using Internet Protocol (IP), the need for communication between programs running on two different computers on the network increased. Such a requirement led to the development of a new type of network sockets that could be used to communicate between two processes running on two separate computers. Therefore, a new address family, AF_INET, was created. The growth of the Internet has made AF_INET the most commonly used address family. The AF_INET family supports protocols such as TCP and UDP.

The type argument of the socket method defines the network connection supported by a socket. As discussed earlier, the network connection can be connection-oriented or connectionless. To create connection-oriented sockets, you use SOCK_STREAM as the `type` value. Connection-oriented sockets, also called stream sockets, are implemented by protocols such as TCP. They are also known as TCP sockets.

To create a connectionless socket, you use SOCK_DGRAM as the `type` value. Connectionless sockets, also called datagram sockets, are implemented by protocols such as UDP. They are also known as UDP sockets.

The other values that can be used for the type argument can be SOCK_RAW, SOCK_RDM, and SOCK_SEQPACKET. Out of all the types discussed, only SOCK_STREAM and SOCK_DGRAM are generally used.

The last argument of the socket method, `protocol`, is optional. It is used with the raw type of sockets and defines the protocol being used. By default, the value of this argument is 0 for all socket types other than raw.

For example, you can create a TCP socket as follows:

```
TCP_Sock=socket(AF_INET, SOCK_STREAM)
```

You can create a UDP socket as follows:

```
TCP_Sock=socket(AF_INET, SOCK_DGRAM)
```

After you create a socket object, you can use it to call various methods. Table 12.2 describes some of these methods.

Table 12.2 Socket Object Methods

METHOD	DESCRIPTION
`accept()`	The `accept()` method accepts a connection and returns the new socket object used to carry out transactions on the connection. The method also returns the address of the socket on the other end of the connection. Before you accept a connection, the socket must be bound to a port and ready to receive connections.
`bind()`	The `bind()` method binds a socket to an address.
`close()`	The `close()` method closes a socket. After the socket is closed, no action can be performed on the socket object.
`connect(address)`	The `connect()` method connects to a socket at a given address.
`getpeername()`	The `getpeername()` method returns the address to which the socket is connected.
`getsockname()`	The `getsockname()` method returns the address of its own socket.
`listen(con_queue)`	The `listen()` method starts listening to requests for connections. This method takes one argument, which is the number of maximum connections that can be queued, before the socket starts refusing them. The number of connections supported by a socket depends on your system, but it has to be at least one.
`makefile(mode,buffer)`	The `makefile()` method creates and returns a file object associated with a socket. This file object can be used to work with file functions, such as `read()` and `write()`. The `makefile()` function takes two arguments. The first argument is the mode of the file object, while the second argument is the buffer size for that object. These arguments are similar in meaning to the arguments of the `open()` built-in function. Only sockets of the address family AF_INET support this function.

METHOD	DESCRIPTION
`recv(buffer, flag)`	The `recv()` method receives data from the socket and returns it as a string. It can take two arguments. The first argument is the buffer size, which limits the maximum size of data that it can receive. The second argument, `flag`, is optional and contains values that are used to perform some advance functions on data. By default, the value of flag is 0.
`recvfrom(buffer, flag)`	The `recvfrom()` method receives data from the socket and returns two values. The first value is the string of data received, and the second value is the address of the sender. This method can take two arguments. The first argument is buffer size, which limits the maximum size of data that it can receive, and the second argument is flag, which has the same meaning as described for `recv()`.
`send(string, flag)`	The `send()` method sends a data string to a socket and returns the size of the data sent. The connection should already exist with a remote socket before you use this function. The meaning of an optional flag argument is the same as that of `recv()`.
`sendto(string, flag, address)`	The `sendto()` method sends a data string to a socket whose address is passed as an argument. Because the address of the remote socket is passed with the function, this function does not require a prior connection. The meaning of an optional flag argument is the same as that of `recv()`.
`shutdown(how)`	The `shutdown()` shuts down the connection. It takes 0, 1, or 2 as the argument. If 0 is passed as an argument, the connection stops receiving data. If 1 is passed, the connection stops sending, and if 2 is passed, the connection stops both sending and receiving.

The format of an address used as arguments in these methods depends on the address family. Generally, the address is a tuple and contains the host name, or the IP address, and the port number of a specific socket.

In addition to the `socket()` method, various other attributes are available in the `socket` module. To use them easily, you can import them in your program by adding this to your code:

```
from module import *
```

Creating a TCP Server and a TCP Client

Now that you have learned to create sockets and learned about their common methods, let's use this knowledge to create servers and clients. As sockets are commonly used for TCP and UDP connections, you will learn to create servers and clients for both these protocols. Let's start with the TCP server.

TCP Server

When creating a TCP server, the server application needs to follow a sequence of steps. The first step in the sequence is to create a socket. To do this, you use the `socket()` method of the `socket` module. After the socket is created, you need to bind the socket to the local computer on which the server is running and assign a unique port number. The host name and the port number together form the address of the socket. This address is then bound to the socket. To do this, you use the `bind()` socket object method. After you have bound the address to the socket, the socket can start listening to the bound port for client requests. For this, you use the `listen()` socket object method. You pass the maximum number of connections that a server can accept as the attribute to the `listen()` method. After the server is ready and listening, it can accept client requests. To do so, you use the `accept()` socket object method. As discussed earlier, when the server accepts the client request, the connection is transferred to a new temporary port. Therefore, the main port is free and open to receive new connections.

Generally, servers are designed to listen for connections indefinitely. To implement this functionality, after a server starts listening for connections, an infinite loop is started. The steps for accepting client connections and other steps for transacting over the connection are included in the loop.

The infinite loop is not meant to end so that the socket always remains open. It is generally a good practice, though, to have a statement to close the socket. To do this, you use the `close()` socket object method. Adding a step to close the socket is a good programming practice and is useful in case a server shuts down unexpectedly.

Let's write the code to create a TCP server:

```
from socket import *

Hostname = ''
PortNumber = 12345
Buffer = 500
ServerAddress = (Hostname, PortNumber)

```

```
8     TCP_Server_Socket = socket(AF_INET, SOCK_STREAM)
9     TCP_Server_Socket.bind(ServerAddress)
10    TCP_Server_Socket.listen(2)
11
12    while 1:
13        print 'Server is waiting for connection'
14        TCP_Client_Socket, ClientAddress = TCP_Server_Socket.accept()
15        print 'Server has accepted the connection request from ',
                    ClientAddress
16        print 'The Server is ready to receive data from the client'
17
18        while 1:
19            ClientData = TCP_Client_Socket.recv(Buffer)
20            if not ClientData:
21                print 'The client has closed the connection'
22                break
23            print 'The client has sent this data string: ',\
                   ClientData
24            TCP_Client_Socket.send('Hello! Client')
25            print 'The server is ready to receive more data
       from the client'
26        TCP_Client_Socket.close()
27
28    TCP_Server_Socket.close()
```

Let's look at this code line by line to understand what is happening:

- In line 1, all attributes of the socket module, including the `socket()` function, are imported.
- In lines 3 through 5, variables are defined for the host name and port number of the server and the maximum size of data that can be exchanged. The `Hostname` variable is left blank, indicating that any available address can be used.
- In line 6, an attribute, `ServerAddress`, is defined. This attribute contains the address of the server. The address consists of the host name and the port number of the server.
- In line 8, the server socket is created and its object, `TCP_Server_Socket`, is returned. The arguments of the `socket()` module denote that the server socket belongs to the address family AF_INET and is a stream socket, SOCK_STREAM.
- In line 9, the address of the server, which consists of the host name and port number, is bound to the socket created in line 8.
- In line 10, the `listen()` method is used to make the socket start listening for connections. The value passed to the `listen()` method denotes that the server can accept a maximum of two incoming connections.
- In line 12, an infinite loop is started, so that the server always listens for client requests.
- In line 14, the client request for a connection is accepted, and the connection is transferred to a new temporary socket, `TCP_Temp_Socket`. The `accept()` method also returns the address of the client.

- In line 18, a new loop that will be used for receiving and sending operations is started.
- In line 19, the data from the client, which has the maximum size equal to the value passed to the `recv()` method, is returned and stored in the attribute `ClientData`.
- In line 20, the if condition checks whether the server has received any data from the client. If the client has sent no data, then it means that the client has quit and the loop has started in line 18. The control shifts to line 26.
- If the server receives data from the client, the control is shifted to line 23.
- In line 23, a message and the data string received from the client are printed.
- In line 24, a data string is send to the client.
- In line 26, the temporary socket created for communication between the client and the server is closed.
- In line 28, the server socket is closed, but due to the infinite loop started in line 12, the control never reaches line 28. Therefore, the preceding code never ends, and the server socket keeps listening for connections indefinitely. If the server shuts down due to some reason, the statement in line 28 will be executed and the server socket will be closed.

As discussed earlier, the server should be running before the client tries to connect to it. We have just created the TCP server; let's now create the TCP client that will connect to this TCP server.

TCP Client

Clients are easier to create than servers. After the server has been started, the client only needs to connect to the server and transact. When creating a client, you need to follow two main steps. First, you need to create a client socket. To do this, you use the `socket()` method of the `socket` module. Second, you need to contact the server to open a connection. For this, you use the `connect()` socket object method and pass the address of the server as an argument to the method.

As discussed earlier, when the server accepts the client request and the connection is established from the client to the server, the server port transfers the connection to a new temporary port. The client, however, is unaware of this and is not concerned with what's happening at the server end. It requires only a connection to the server. Now that the connection has been established, the transaction of sending and receiving can happen between both the client and the server. The client remains connected to the server only until the time it wants to transact. As soon as the transaction is completed, the client closes its socket and ends the connection to the server. This shows that the client connection is transaction-specific, while the server on the other end keeps listening for connections indefinitely.

Let's write the code to create a TCP client:

```
from socket import *

Hostname = 'localhost'
PortNumber = 12345
```

```
5     Buffer = 500
6     ServerAddress = (Hostname, PortNumber)
7
8     TCP_Client_Socket = socket(AF_INET, SOCK_STREAM)
9     TCP_Client_Socket.connect(ServerAddress)
10
11     while 1:
12         print 'The client is connected to the server'
13         DataStr = raw_input('Enter data to send to the server: ')
14         if not DataStr:
15             print 'The client has entered nothing; hence the
        connection to the server is closed'
16             break
17         TCP_Client_Socket.send(DataStr)
18         ServerData = TCP_Client_Socket.recv(Buffer)
19         if not ServerData:
20             print 'The server has sent nothing'
21             break
22         print 'The server has sent this data string: ', ServerData
23     TCP_Client_Socket.close()
```

Let's look at this code line by line to understand what is happening:

- In line 1, all attributes of the socket module, including the `socket()` function, are imported.
- In lines 3 through 5, variables are defined for the host name and port number of the server and the maximum size of data that can be exchanged. The Hostname variable will contain the name of the host on which the server is running. In this case, both the server and the client are running on the same host. Therefore, the value `localhost` is passed as the host name.
- In line 6, an attribute, `ServerAddress`, that contains the address of the server, is defined. The address consists of the host name and port number of the server.
- In line 8, the client socket is created and its object, `TCP_Client_Socket`, is returned. The arguments of the `socket()` module denote that the client socket belongs to the address family AF_INET. The arguments also denote that the client socket is a stream socket, SOCK_STREAM.
- In line 9, the `connect()` method is used to connect the client to the server. The address of the server, which consists of the host name and the port number, is passed as an argument to the `connect()` method.
- In line 11, a loop contains statement is started. This statement will be used for sending and receiving operations.
- In line 13, the user at the client end is prompted for data input. The data string is stored in the attribute `DataStr`.
- In line 14, the `if` condition checks whether the user at the client end has entered any data. If not, the loop started in line 11 breaks and the control shifts to line 23.
- If the user at the client end enters data, the control shifts from line 14 to line 17.

- In line 17, the data entered by the user at the client end is sent to the server to which the client is connected.
- In line 18, the data from the server, which has the maximum size equal to the value passed to the `recv()` method, is returned and stored in the attribute `ServerData`.
- In line 19, the `if` condition checks whether the server has sent any data to the client. If not, the loop started in line 11 breaks and the control shifts to line 23.
- If the server sends data, the control shifts from line 14 to line 22.
- In line 22, a message and the data received from the server are printed.
- In line 23, the client socket is closed.

Executing the TCP Server and the TCP Client

The code for both TCP server and the client are ready. Let's run them and see how client/server software architecture actually works.

You have to be careful about the sequence in which you run these applications. As discussed earlier, the server should be running before the client tries to connect to it. Therefore, the server application has to be run first. After the server has started, the client application is run.

In order to execute the code written for the TCP server and the client, we save them as a Python file with an extension `.py`. The server program is saved as `TCPserver.py`, and the client program is saved as `TCPclient.py`. In this case, both the server and the client will run on the same host. Therefore, to show the working of both the server side and the client side, you will have to run `TCPserver.py` in one terminal window and `TCPclient.py` in another terminal window.

Let's execute the TCP server program.

```
$ python TCPserver.py
Server is waiting for connection
```

The server is running and listening to its port for connections. Now let's execute the TCP client program and connect to the server.

```
$ python TCPclient.py
The client is connected to the server
Enter data to send to the server:
```

After the client connects to the server, the server shows the following message:

```
.....
.....
The server has accepted the connection request from ('127.0.0.1, 1131)
The server is ready to receive data from the client
```

NOTE In the preceding code, the port number is randomly specified by the system and can be different every time.

Now that the client and the server are connected and the server is ready to receive data from the client, let's enter data at the client end say, `Hello! Server`.

```
.....
.....
Enter data to send to the server: Hello! Server
```

After the client sends the data to the server, the server displays a message and is ready to receive more data from the client.

```
.....
.....
The client has sent this data string: Hello! Server
The server is ready to receive more data from the client
```

When the server receives the data from the client, the server sends a message to the client and the client is ready to send more data.

```
.....
.....
The server has sent this data string: Hello! Client
The client is connected to the server
Enter data to send to the server:
```

You can exchange more data between the client and the server as shown here. If you want to close the client connection with the server, you need to send a blank string. You can do this by pressing the Enter key when the system prompts you for input.

```
.....
.....
Enter data to send to the server:
The user has entered nothing; hence the connection to the server is
closed
$
```

After the client sends a blank string to the server, the server displays a message and closes the temporary socket created by it for this connection. Even if this connection is closed, the server is running and listening for more connections.

```
.....
.....
The client has closed the connection
Server is waiting for connection
```

Creating a UDP Server and a UDP Client

In the previous sections, you learned to write and execute programs for a TCP server and a TCP client. Let's now create a UDP server and a UDP client taking the same example for creating a TCP server and a client. This will help you understand the

difference between programming for a connection-oriented (TCP) and connectionless (UDP) type of network environment.

UDP is connectionless and does not try to establish a connection before sending and receiving data. Due to this nature of UDP, you cannot be sure whether the other side has received the data. Let's now create a UDP server first.

UDP Server

There is some difference in the steps that you follow for creating a TCP server and a UDP server. UDP is not connection-oriented; therefore, the amount of setup required for a UDP server is less than that required for a TCP server.

When creating a UDP server, the server application needs to follow a sequence of steps, starting with creating a socket. To do this, you use the `socket()` method of the `socket` module. After the socket is created, you need to bind the address to the socket. For this, you use the `bind()` socket object method. After you have bound the address to the socket, the UDP server can query the port for client connections.

These are all the steps required for making the UDP server ready to receive client connections. As the UDP server is connectionless, the number of subsequent connections need not be specified using the `listen()` socket object method for UDP servers. For the same reason, after the UDP server accepts the client connection, the connection is not transferred to a new temporary socket. Therefore, the UDP server does not require `accept()`, `recv()`, and `send()` socket object methods. Instead, the UDP server uses the `recvfrom()` and `sendto()` socket object methods for receiving and sending transactions.

Servers generally run indefinitely; therefore, you can also make the UDP server run indefinitely by using an infinite loop. This loop is not meant to end so that the server always queries the port for new connections. As discussed earlier, though, it is a good practice to include a statement to close the server socket.

Let's write the code to create a UDP server:

```
from socket import *

Hostname=''
PortNumber=12345
Buffer=500
ServerAddress=(Hostname, PortNumber)

UDP_Server_Socket=socket(AF_INET, SOCK_DGRAM)
UDP_Server_Socket.bind(ServerAddress)

while 1:
    print 'The server is ready to receive data from the client'
    ClientData, ClientAddress = UDP_Server_Socket.recvfrom(Buffer)
    print 'Server has received data from ', ClientAddress
    print 'The client has send this data string: ', ClientData
    UDP_Server_Socket.sendto('Hello! Client', ClientAddress)
UDP_Server_Socket.close()
```

Let's look at this code line by line to understand what is happening. It is somewhat similar to the code used for the TCP server.

- In line 1, all attributes of the socket module, including the `socket()` function, are imported.
- In lines 3 through 5, variables are defined for the host name and port number of the server and the maximum size of data that can be exchanged. The `Hostname` variable is left blank, indicating that any available address can be used.
- In line 6, an attribute, `ServerAddress`, containing the address of the server is defined. The address consists of the host name and port number of the server.
- In line 8, the server socket is created and its object, `UDP_Server_Socket`, is returned. The arguments of the `socket()` module denote that the server socket belongs to the address family AF_INET. The arguments also indicate that the server socket is a stream socket, `SOCK_DGRAM`.
- In line 9, the address of the server, which consists of the host name and the port number, is bound to the socket created in line 8.
- In line 11, an infinite loop is started.
- In line 13, the `recvfrom()` stock object method returns the data from the client and the address of the client. The data received is stored in the attribute `ClientData`, and the address of the client is stored in the attribute `ClientAddress`.
- In line 14, a message and the client address are printed.
- In line 15, a message and the data string received from the client are printed.
- In line 16, a data string is sent to the client.
- In line 17, the server socket is closed, but due to the infinite loop started in line 11, the control never reaches line 17. If the server shuts down for some reason, the statement in line 17 will be executed and the server socket will be closed.

We have just created the UDP server; let's now create the UDP client, which will connect to this UDP server.

UDP Client

Because UDP clients are connectionless, the code used for a UDP client is a little different from that of the TCP client. In the case of a UDP client, data from the server needs to be sent or received only. When creating a UDP client, you need to create a client socket. To do this, you use the `socket()` method of the `socket` module. Because UDP is connectionless, you do not need to open a separate connection with the server to exchange data between the UDP client and the server. Therefore, there is no need for the `connect()`, `send()`, and `recv()` socket object methods. Instead, you use the `sendto()` and `recvfrom()` socket object methods for receiving and sending transactions.

The client connects to the server only until the time a transaction such as sending or receiving data is taking place. Let's write the code to create a UDP client.

```
1     from socket import *
2
3     Hostname='localhost'
4     PortNumber=12345
5     Buffer=500
6     ServerAddress=(Hostname, PortNumber)
7
8     UDP_Client_Socket=socket(AF_INET, SOCK_DGRAM)
9
10     while 1:
11        DataStr=raw_input('Enter data to send to the server: ')
12        if not DataStr:
13            print 'The user has entered nothing; hence the client
                 socket is closed'
14            break
15        UDP_Client_Socket.sendto(DataStr, ServerAddress)
16        ServerData, ServerAddress = UDP_Client_Socket.recvfrom(Buffer)
17        if not ServerData:
18            print 'The server has sent nothing; hence the client
                 socket is closed'
19            break
20        print 'The server has sent this data string: ', ServerData
21     UDP_Client_Socket.close()
```

Let's look at this code line by line to understand what is happening. It is quite similar to the one used for TCP client.

- In line 1, all attributes of the socket module, including the `socket()` function, are imported.
- In lines 3 through 5, variables are defined for the host name and port number of the server and the maximum size of data that can be exchanged. The `Hostname` variable will contain the name of the host on which the server is running. In this case, both the server and the client are running on the same host. Therefore, the value `localhost` is passed as the host name.
- In line 6, an attribute, `ServerAddress`, containing the address of the server is defined. The address consists of the host name and port number of the server.
- In line 8, the server socket is created and its object, `UDP_Client_Socket`, is returned. The arguments of the `socket()` module denote that the server socket belongs to the address family AF_INET. The argument also denotes that the server socket is a stream socket, `SOCK_DGRAM`.
- In line 10, a loop containing statements is started. This statement will be used for sending and receiving operations.
- In line 11, the user at the client end is prompted for data input. The data string is stored in the attribute `DataStr`.
- In line 12, the `if` condition checks whether the user at the client end has entered any data. If not, then the loop started in line 10 breaks and the control shifts to line 21.

- If the user at the client end enters data, the control shifts from line 12 to line 15.
- In line 15, the data entered by a user at the client end is sent to the server whose address is passed as an argument in the `sendto()` socket object method.
- In line 16, the `recvfrom()` stock object method returns the data from the server and the address of the server. The data received is stored in the attribute `ServerData`, and the address of the client is stored in the attribute `ServerAddress`.
- In line 17, the `if` condition checks whether the server has sent any data to the client. If not, the loop started in line 10 breaks and the control shifts to line 21.
- If the server sends data, the control shifts from line 17 to line 20.
- In line 20, a message and the data received from the server are printed.
- In line 21, the client socket is closed.

Executing the UDP Server and the UDP Client

As with any type of server and client, a server should be running before a client tries to connect to it. Therefore, the server application has to be run first. After the server has started, the client application is run.

For the purpose of executing the code written for the UDP server and the UDP client, we save them as a Python file with an extension `.py`. The server program is saved as `UDPserver.py`, and the client program is saved as `UDPclient.py`. In this case, both the server and the client will run on the same host. Therefore, to show the working of both the server side and the client side, you will have to run `UDPserver.py` in one terminal window and `UDPclient.py` in another terminal window.

```
$ python UDPserver.py
The server is ready to receive data from the client
```

The server is running and ready to receive data from the client. Now let's execute the UDP client program and send some data to the server.

```
$ python UDPclient.py
Enter data to send to the server: Hello! Server
```

After the client sends the data to the server, the server displays some messages and is ready to receive more data from the client.

```
.....
.....
Server has received data from ('127.0.0.1', 1028)
The client has sent this data string: Hello! Server
The server is ready to receive data from the client
```

In this code, the port number is randomly given by the system and can be different every time. When the server receives the data from the client, the server sends a message to the client, and the client is ready to send more data.

```
.....
.....
The server has sent this data string: Hello! Client
Enter data to send to the server:
```

You can exchange more data between the client and the server as shown here. If you want to close the client socket, you need to send a blank string. You can do this by pressing the `Enter` key when the system prompts you for input.

```
.....
.....
Enter data to send to the server:
The user has entered nothing; hence the client socket is closed
$
```

These sections described the process of creating and executing TCP and UCP, servers and clients. You were also able to differentiate between the processes followed for TCP and UDP. Note that the difference between the process of creating codes for TCP and UDP is also seen in the messages used for them.

The IT department computer is the main computer and will therefore store data. In other words, the IT department computer serves as the server while the Admissions office computer serves as the client. One of the major concerns of the university management team is the integrity of the data exchanged. Therefore, the socket used should be connection oriented. The University network is IP-based, and therefore it supports connection-oriented protocols, such as TCP. Both server and client sockets are of the address family AF_INET and type SOCK_STREAM.

```
Server socket (IT Department computer)
Server_Socket=socket(AF_INET, SOCK_STREAM)
Client socket (Admission Office computer
Client_Socket=socket(AF_INET, SOCK_STREAM)
```

Other Network Programming-Related Modules

Python provides many other modules that are used for implementing some advanced feature of network programming. Some of these modules are briefly described here.

`asyncore`

The `asyncore` module is used to write and handle servers and clients that use asynchronous socket service. It keeps a check on the sockets to find information on the data transfer that is taking place with them. Based on the situation the `asyncore` module handles them by implementing an appropriate routine.

An important part of the `asyncore` module is the `dispatcher` class. You use the `dispatcher` class by creating subclasses and overriding the required method. It defines various methods to handle different situations, such as `handle_write()`, `handle_read()`, `handle_connect()`, and `handle_close()`. The `dispatcher` class also wraps the `socket` object.

select

The `select` module is generally available in many operating systems. Windows uses this module only for sockets. With other operating systems, such as Unix, the module is also used for pipes and files. The `asyncore` module, as discussed earlier, is built on top of this module.

This module provides `select()` and `poll()` functions. The `select()` function takes three arguments. These three arguments are socket lists and are used for input, output, and exceptional conditions. The function can have one optional argument, which provides time in seconds as time-out. It returns three lists of ready objects, which are the subsets of the first three arguments passed to the function. If no file descriptor is ready before time-out, the three lists are returned empty.

The `poll()` function returns a polling object. This object is used for registering a file descriptor and also for unregistering or removing a registered file descriptor. These are then used for polling I/O events. The `poll()` function is not supported by all operating systems.

Write the Code to Run on the IT Department Computer

The code for the server (IT Department Computer) is as follows:

```
from socket import *    #Imports the attributes of the socket module
Hostname = ''       #Defines the host name/IP address of the server.
#The variable is left blank, so that any available address can be used
PortNumber = 22222  #Defines a dedicated port number for the server
Buffer = 1024       #Defines the maximum size of data that can be
exchanged
ServerAddress = (Hostname, PortNumber) #Defines the address of the
server
Server_Socket=socket(AF_INET, SOCK_STREAM)      #Creates a stream
                                             #socket for the server
Server_Socket.bind(ServerAddress)       #Binds the server address to
                                        #the server socket
Server_Socket.listen(5)     #Listens for connections
print
print '!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'
print 'Server is waiting for connection'
print '!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'
print
while 1:        #Infinite loop starts
    Temp_Socket, ClientAddress = Server_Socket.accept()     #Accepts
client connection and passes it to a new temporary socket
    print 'Server has accepted the connection request from ',
ClientAddress
    print
    print 'The server is ready to receive data from the client'
    print '^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^'
    print

    while 1:        #Client/server connection loop starts
```

```
        DataFromClient = Temp_Socket.recv(Buffer)       #Receives data
                                                        #from the client
        if not DataFromClient:      #Checks if the variable is blank
            print
            print
            print '************************************'
            print 'The client has closed the connection'
            print '************************************'
            print
            print
            print
            print
            print '!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'
            print 'Server is waiting for a new connection'
            print '!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'
            print
            break       #Client/server connection loop breaks
        WriteToFile=open('DataFile', 'a')     #Opens a file in append
                                              #mode
        WriteToFile.write(DataFromClient + '\n')     #Writes data to
                                                     #file
        WriteToFile.close()     #Closes the file
        ReadfromFile=open('DataFile','r')
        output=ReadfromFile.read()
        Temp_Socket.send('DATA \n%s \nWRITTEN TO THE FILE' % output)
        #Sends data to the client
        ReadfromFile.close()
        print
        print '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
        print 'The server is ready to receive more data from the client'
        print '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    Temp_Socket.close()     #Closes the temporary socket
Server_Socket.close()       #Closes the server socket and stops the
                            #infinite loop
```

Write the Code to Run on the Admission Office Computer

The code for the client (Admission Office Computer) is as follows:

```
from socket import *        #Imports the attributes of the socket module
Hostname = '172.17.68.120'      #Defines the IP address of the server
#(IT Department computer). Uses the host name/IP address of the
#computer on which you are executing the server
PortNumber = 22222      #Defines the dedicated port number of the server
```

```
Buffer = 1024       #Defines the maximum size of data that can be
exchanged
ServerAddress = (Hostname, PortNumber)   #Defines the address of server
Client_Socket=socket(AF_INET, SOCK_STREAM)      #Creates a stream
                                            #socket for the client
Client_Socket.connect(ServerAddress)        #Connects to the server
                                            #at a given address
print
print 'The client is connected to the server at', ServerAddress
print
while 1:
    DataToServer = raw_input('Enter data: ')        #Asks input for data
    if not Data:        #Checks if the variable is blank
        print
        print '********************************************'
        print '**        You have entered nothing        **'
        print '** The connection to the server is closed **'
        print '********************************************'
        print
        break
    Client_Socket.send(DataToServer)        #Sends data to server
    ServerData=Client_Socket.recv(Buffer)    #Receives data from client
    if not ServerData:      #Checks if the variable is blank
        print
        print 'The server has ended the connection'
        break
    print '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    print 'Message from the server:', ServerData
    print '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    print
Client_Socket.close()       #Closes the client socket
```

Execute the Code Created for the IT Department Computer

To be able to implement or view the output of the server programs, perform the following steps on the server computer (the IT department computer):

1. Write the server code in a text editor and save it with the `.py` extension.
2. Open a terminal window.
3. At the shell prompt, type `python` followed by the name of the server file, if the file is in the current directory. The server starts as shown in Figure 12.1.

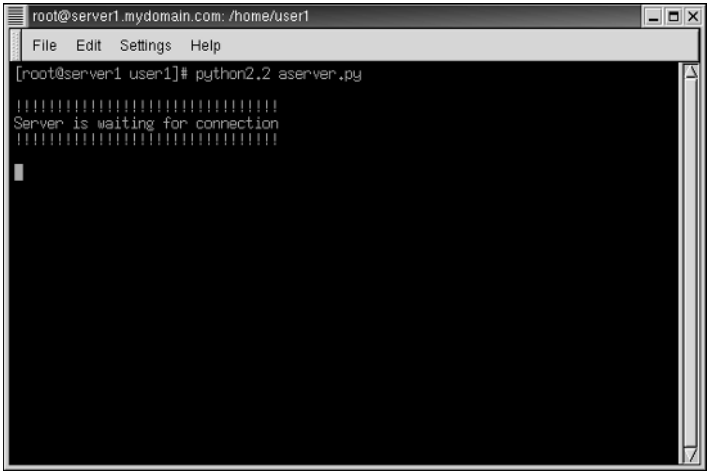


Figure 12.1 The screen after starting the server.

Execute the Code Created for the Admission Office Computer

To be able to implement or view the output of the server programs, perform the following steps on the client computer (Admissions office computer):

1. Write the client code in a text editor and save it with the .py extension.
2. Open a new terminal window.
3. At the shell prompt, type `python` followed by the name of the client file, if the file is in the current directory.
4. At the prompt `Enter data:`, enter `Hello! Server` as shown in Figure 12.2.

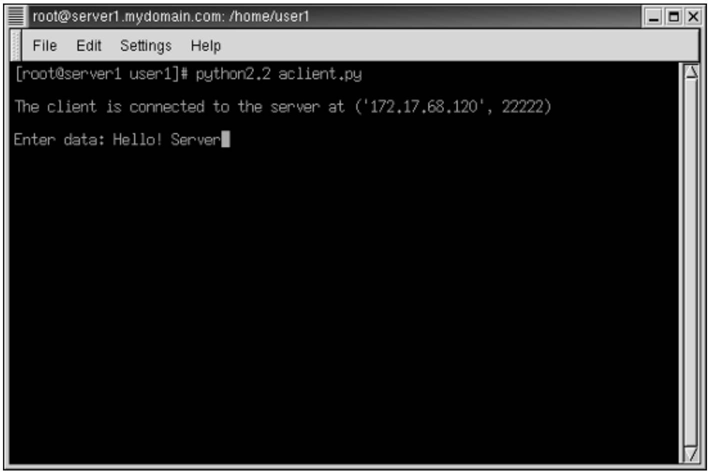


Figure 12.2 The client sending data to the server.

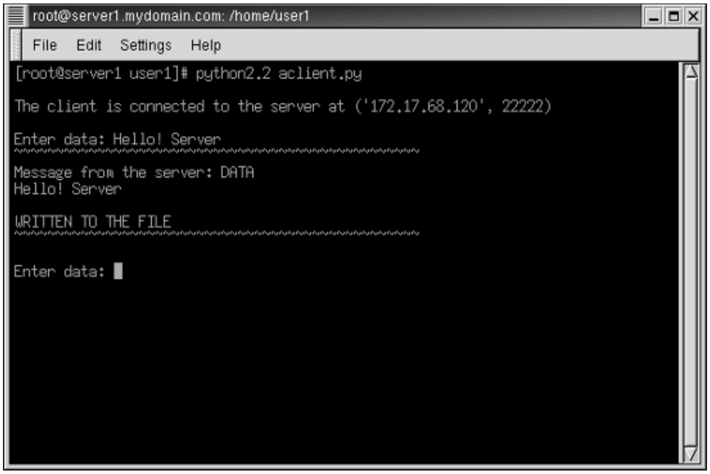


Figure 12.3 The server sending data to the client.

5. The server sends the data written to the file back to the client as shown in Figure 12.3.
6. At the prompt `Enter data:`, enter You are too fast and stable! as shown in Figure 12.4.
7. Once again, the server sends the data written to the file back to the client, as shown in Figure 12.5.
8. In order to exit from the client program, at the prompt `Enter data:`, press the Enter key.

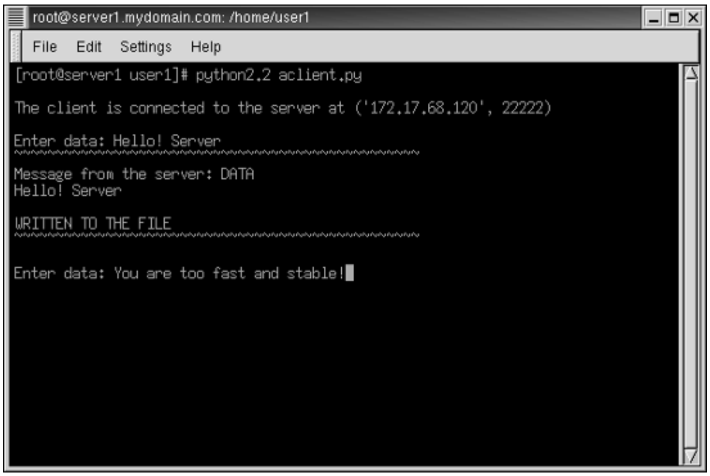


Figure 12.4 The client sending data to the server again.

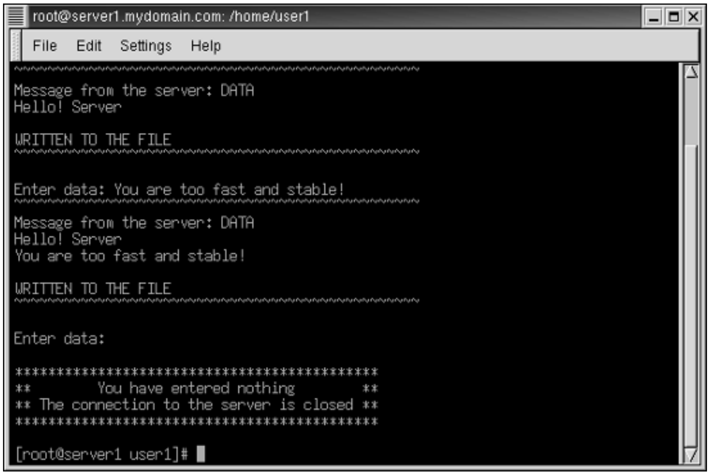


Figure 12.5 The server sending data to the client again.

NOTE Regularly observe the output of both programs to understand what is happening.

The server program will not end on its own because it is in an infinite loop. You will have to close the terminal window in which the server program is running in order to end it.

Verify That Data Has Been Saved to a File in the IT Department Computer

To verify that all data has been saved to a file, perform the following steps on the server computer (the IT department computer):

1. Open the file `DataFile` from the same directory where your server program is saved.
2. Observe the contents of the file. It displays the complete data that you have entered on the client end.
3. Close the file.

Summary

In this chapter, you learned the following:

- *Client/server architecture* is a term used to define the communication process between two computers, one of which is a client and the other is the server.

- The server provides services that one or more clients require.
- A successful transaction in client/server architecture begins when the client asks a server for some service and ends after the server replies to the client request.
- The server should always be ready to receive client requests.
- Client/server architecture applies to both:
  - Hardware
  - Software
- In a software client/server architecture system, the client and the server are not hardware devices but programs running on some hardware.
- The rules used for network communication are called network protocols.
- Protocols can be of two types:
  - Connection-oriented, such as TCP
  - Connectionless, such as UDP
- Python supports both types of protocols.
- Sockets are software objects that allow programs to make connections.
- Sockets are the only IPC forms that support cross-platform communication.
- Sockets are bound to ports, which are numeric addresses.
- A few port numbers are assigned to specific protocols.
- The client should know the host name, or the IP address, of the server computer and the port number for the client to connect to the server.
- To implement network programming in Python by using sockets, you use the `socket` module.
- The `socket()` method of the `socket` module is used to create a new socket.
- The syntax of the `socket()` method is as follows:

```
socket(family, type, protocol)
```

- Python supports two address families:
  - AF_UNIX
  - AF_INET
- These family names define whether the client and server programs run on the same or different computers.
- The sockets of the AF_UNIX family are also called Unix sockets.
- AF_INET is the most commonly used address family and supports protocols such as TCP and UDP.
- Sockets can be of the types:
  - SOCK_STREAM
  - SOCK_DGRAM

  - SOCK_RAW
  - SOCK_RDM
  - SOCK_SEQPACKET
- SOCK_STREAM is used to create connection-oriented sockets.
- SOCK_STREAM is used to create connectionless sockets.
- The socket object can be used to execute the following methods:
  - `accept()`
  - `bind()`
  - `close()`
  - `connect()`
  - `getpeername()`
  - `getsockname()`
  - `listen()`
  - `makefile()`
  - `recv()`
  - `recvfrom()`
  - `send()`
  - `sendto()`
  - `shutdown()`
- Socket object methods, such as `accept()`, `listen()`, `recv()`, and `send()`, are used for working with connection-oriented servers and clients.
- Socket object methods, such as `recvfrom()` and `sendto()`, are used for working with connection-oriented servers and clients.
- The `asyncore` module is used to write and handle servers and clients that use asynchronous socket service.
- The `dispatcher` class is an important part of the `asyncore` module.
- The `asyncore` module, as discussed earlier, is built on top of the `select` module.
- The select module provides two functions:
  - `select()`
  - `poll()`
- The `SocketServer` module is used for creating general IP servers, provides the necessary framework for network servers, and simplifies the job of writing them.
- The `BaseHTTPServer` module provides the infrastructure required for creating Web servers.

- The `BaseHTTPServer` module defines two classes:
  - `HTTPServer`
  - `BaseHTTPRequestHandler`
- The `HTTPServer` class is used to create Web sockets.
- The `BaseHTTPRequestHandler` handles HTTP requests.
- The `SimpleHTTPServer` module is used for creating simple Web servers.
- The `SimpleHTTPServer` module defines a class, `SimpleHTTPRequestHandler`, for handling requests to serve only base directory files.
- The `CGIHTTPServer` module is used for creating Web servers with support for CGI.
- The `CGIHTTPServer` module defines a class, `CGIHTTPRequestHandler`, for handling requests to serve files or output of CGI scripts.

CHAPTER 13

Multithreaded Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Identify the difference between a single-threaded application and a multithreaded application**
- **Create threads by using the `thread` module**
- **Create multithreaded applications by using the `threading` module**

Getting Started

Graphics and sound are part of most Web-based training courses. Have you ever noticed that graphics, text, and audio effects run simultaneously? Imagine a situation where the screen changes, then text appears. Next, the audio starts playing. To use a course to our advantage, all the elements of the course need to be processed at the same time. In other words, the application for the course needs to be divided into three subunits. Each subunit handles one task.

In a similar manner, suppose a Web server could talk only to a single user at a time. With so many people wanting to access the same Web site, you may end up waiting a very long time to make it to the front of the queue!

Fortunately, the courses on the Web and the Web servers are designed to perform many things at the same time. Each of these techniques involves splitting a problem into smaller tasks and allowing them to run almost at the same time. (Obviously, they can't truly run at the same time unless your computer has more than one processor.)

A *thread*, which is the smallest unit of code that can be executed, performs each of these tasks. Any program that has more than one thread is called a *multithreaded program*. Having looked at a thread, it is essential for us to look at another component to complete the big picture here—a *process*. A process is an executing instance of a program. This raises the question about how a process is different from a thread. The essential difference is that a process has a complete set of data and variables while one or more threads may share the same data.

The primary advantage of creating multithreaded applications is that you can write efficient programs that make maximum use of the processor by keeping idle time to a minimum. Before you can design multithreaded applications, you first have to understand the difference between a single-threaded application and a multithreaded application. To do this, let's first examine a single-threaded application.

Single-Threaded Applications

A single-threaded application has only one thread. This thread is the main thread of the application. In this application, all the processing is done in a linear fashion. In other words, user input and any processing that does not require user input are handled by the same thread. While waiting for user input, the application cannot perform any background task due to the presence of only one thread. Therefore, the application is not able to effectively switch between various independent tasks. A single-threaded application also takes a long time to execute.

To better understand this, let's consider an example. The program **singlethr.py** uses a single-threaded application. This program contains two functions and executes them one after the other.

```
from time import sleep,ctime,time
def func1():
    i=0
    while i<=3:
        print "func1 at", ctime(time())
        sleep(2)
        i=i+1
def func2():
    j=0
    while j<=3:
        print "func2 at", ctime(time())
        sleep(1)
        j=j+1
print '*'*40
print "started at", ctime(time())
print '*'*40
func1()
print '*'*40
func2()
```

```
print '*'*40
print "end at", ctime(time())
print '*'*40
```

In the preceding code, we created two functions, `func1()` and `func2()`. In addition, we used the `sleep()` function of the time module. This function takes the number of seconds as an argument in the form of a floating-point value and halts the execution for the specified number of seconds. The definition of `func1()` contains a call to the `sleep()` function with an argument of 2 seconds. Similarly, `func2()` contains a call to the `sleep()` function with an argument of 1 second. This means that when the program execution encounters these function calls, the program execution is halted for a specified number of seconds. The execution of `func1()` completes first before `func2()` starts to execute because `func1()` and `func2()` are called sequentially. The execution of `func1()` takes 8 seconds, and the execution of `func2()` takes 4 seconds. Therefore, the total program execution takes 12 (8 + 4) seconds. The output of the execution of **singlethr.py** is as follows:

```
****************************************
started at Thu Dec 27 19:10:07 2001
****************************************
func1 at Thu Dec 27 19:10:07 2001
func1 at Thu Dec 27 19:10:09 2001
func1 at Thu Dec 27 19:10:11 2001
func1 at Thu Dec 27 19:10:13 2001
****************************************
func2 at Thu Dec 27 19:10:15 2001
func2 at Thu Dec 27 19:10:16 2001
func2 at Thu Dec 27 19:10:17 2001
func2 at Thu Dec 27 19:10:18 2001
****************************************
end at Thu Dec 27 19:10:19 2001
****************************************
```

After considering the disadvantages of a single-threaded application in terms of time, let's scrutinize how Python implements threading.

Threading in Python

Execution of Python applications is controlled by *Python Virtual Machine (PVM)*, which is also known as the Python interpreter's main loop. Python is designed in such a way that only one thread can take control of this main loop at a time. Therefore, even if multiple threads are alive, only one thread can be executed at any given time. A *global interpreter lock (GIL)* controls access to Python Virtual Machine. This lock ensures that exactly one thread is executing in the main loop at a particular point in time. In a multithreaded environment, Python Virtual Machine switches on the GIL for a thread that is to be executed. Next, PVM executes the thread for a specified number of time periods and then makes the thread sleep. Next, it unlocks the GIL and repeats the process for another thread until all the threads have executed completely.

Creating Multithreaded Applications

Problem Statement

Techsity University is converting all its systems to client/server architecture over the IP network. Jenny's team is working to demonstrate the use of client/server architecture and the security of data sent over the network to the management team. Jenny's team has already implemented socket programming to show the working of client/server architecture between the computers in the Admissions office and the IT department. The team is now assigned the task of creating a chat application between the computers of the IT department. To start, this chat application should be able to connect five computers at a time in the IT department. When a client sends a message to the server, the server sends the message back to the client. In addition, when any other client connects to the server, all the earlier messages sent by the existing clients are sent to that client.

Let's identify the tasks for creating this chat application.

Task List

- **Identify the class and the methods required to create a multithreaded application.**
- **Write the code for the server.**
- **Write the code for the client.**
- **Execute the code created for the server.**
- **Execute the code created for the client.**

Identify the Class and the Methods to Create a Multithreaded Application

Python provides two modules to support multithreaded programming. They are the `thread` module and the `threading` module. These modules are used to manage threads. If Python is not configured for threads when it is built, then it basically relies on operating system threads. Otherwise, the `thread` and `threading` modules can be used to create efficient multithreaded applications. The `thread` module provides the basic thread and locking features and is appropriate for lower-level thread access, whereas the `threading` module provides higher-level thread access. Let's examine each of them with the help of examples.

The `thread` Module

The `thread` module provides the functionality of working with threads, including generating threads and locking them. Locking threads is a synchronization mechanism that is accomplished by the use of a *lock object*. Only one thread can acquire a lock at a time. Here are the most commonly used thread functions available in the `thread` module.

thread.start_new_thread(func, args,[,kwargs]). This function starts a new thread and uses an `apply()` function internally to call the function `func` with `args` arguments and `kwargs` optional arguments. `args` should be a tuple.

thread.exit(). This function ends a thread.

thread.get_thread(). This function returns the identifier of the current thread.

thread.allocate_lock(). This function creates and returns a lock object. The following functions are exposed to a lock object:

lock_obj.acquire([waitflag]). This function is used to acquire a lock and takes an optional argument, `waitflag`. If `waitflag` is omitted, this method acquires the lock immediately and returns `None` after acquiring the lock. If the value of the `waitflag` argument is zero, the lock is acquired only when it can be acquired immediately without waiting for another thread. For any other nonzero value of `waitflag`, the lock is acquired immediately after another thread, if in execution, releases the lock. When an argument is supplied, the return value is `1` if the lock is acquired successfully, `0` if not.

lock_obj.release(). This function releases the lock.

lock_obj.locked(). This function returns `1` if a thread acquires a lock. It returns `0` if a thread does not acquire a lock.

Let's change the previous code of `singlethr.py` to incorporate threads and change it to `multithr.py` as follows:

```
from time import sleep,ctime,time
import thread
def func1():
    i=0
    while i<=3:
        print "func1 at", ctime(time())
        sleep(2)
        i=i+1
def func2():
    j=0
    while j<=3:
        print "func2 at", ctime(time())
        sleep(1)
        j=j+1
print '*'*40
print "started at", ctime(time())
print '*'*40
thread.start_new_thread(func1,())
print '*'*40
thread.start_new_thread(func2,())
sleep(9)
print '*'*40
print "end at", ctime(time())
print '*'*40
```

The only change from `singlethr.py` to `multithr.py` is that we have introduced threads in the latter. The first highlighted statement in the preceding code starts a new thread in which `func1()` is invoked. The program execution control immediately moves to the next statement. It encounters another `start_new_thread()` function and therefore starts another thread in which `func2()` is invoked. Again, the program execution moves to the next statement and encounters a `sleep()` function. Why is the call to the `sleep()` function important? If the main thread is not put to sleep, the execution of the main thread will continue. The last print statement will be executed, and the program control will end the main loop while the other threads are still running. Notice the following output of `multithr.py`.

```
****************************************
started at Sat Dec 29 16:39:20 2001
****************************************
****************************************
func1 at Sat Dec 29 16:39:20 2001
func2 at Sat Dec 29 16:39:20 2001
func2 at Sat Dec 29 16:39:21 2001
func1 at Sat Dec 29 16:39:22 2001
func2 at Sat Dec 29 16:39:22 2001
func2 at Sat Dec 29 16:39:23 2001
func1 at Sat Dec 29 16:39:24 2001
func1 at Sat Dec 29 16:39:26 2001
****************************************
end at Sat Dec 29 16:39:29 2001
****************************************
```

Notice that the execution of the thread containing `func1()` starts first and the thread sleeps for two seconds. This is the time when the execution of `func2()` takes place twice because the sleep duration of `func2()` is set to one second. Again, the execution of `func1()` continues while `func2()` is sleeping. The execution continues till both the threads have completed the execution of the functions. Notice the amount of time saved (three seconds) in comparison with the previous single-threaded application, `singlethr.py`. In `singlethr.py`, first the execution of `func2()` could start only after the execution of `func1()` was complete. Here, the execution of both the functions takes place simultaneously and none has to wait for the other to finish execution before the other starts.

To understand lock objects, let's consider yet another example. Consider `lockthr.py`.

```
import thread
from time import ctime,time,sleep
class Bank:
    def __init__(self):
        self._account={}
        self._account['1']=self.account_savings
        self._account['2']=self.account_current
        self._account['3']=self.account_fixed
        self._account['4']=self.account_recrg
    def account(self,selection,seconds):
        self._account[selection](seconds)
```

```
    def account_savings(self,seconds_arg):
        thread.start_new_thread(self.openac,(seconds_arg,\
                                '1. Savings',locks[0]))
    def account_current(self,seconds_arg):
        thread.start_new_thread(self.openac,(seconds_arg,\
                                '2. Current',locks[1]))
    def account_fixed(self,seconds_arg):
        thread.start_new_thread(self.openac,(seconds_arg,\
                                '3. Fixed',locks[2]))
    def account_recrg(self,seconds_arg):
        thread.start_new_thread(self.openac,(seconds_arg,\
                                '4. Recurring',locks[3]))
    def openac(self,seconds,account,lock):
        for i in range(seconds):
            sleep(0.01)
        print "%s is opened at %s" % (account,ctime(time()))
myBank=Bank()
locks=[]
for i in range(4):
    lock=thread.allocate_lock()
    lock.acquire()
    locks.append(lock)
print "start at",ctime(time())
myBank.account('1',700)
myBank.account('2',500)
myBank.account('3',500)
myBank.account('4',300)
```

The preceding code uses threads to open four types of bank accounts. Notice that four threads are used to execute the `openac()` function four times with different types of accounts. In addition to the name of the account that is to be opened, a lock object is passed. Four lock objects are created separately in the main thread by using the `thread.allocate_lock()` function and are acquired using the `acquire()` method. Each of the locks is added to the list `locks` after the locks have been acquired. Each time the `openac()` function is called by the threads, the calling function passes a lock as an argument along with the number of seconds and the account type. In the preceding code, all lock objects are acquired first because acquiring locks is time-consuming. If the threads execute quickly, it is possible that they complete execution even before the lock is acquired. In addition, this technique enables all the threads to start at nearly the same time. The output of the code will be this:

```
start at Mon Dec 31 11:15:03 2001
4. Recurring is opened at Mon Dec 31 11:15:06 2001
2. Current is opened at Mon Dec 31 11:15:08 2001
3. Fixed is opened at Mon Dec 31 11:15:08 2001
1. Savings is opened at Mon Dec 31 11:15:10 2001
```

As stated earlier, the `thread` module provides the basic support for threading while the `threading` module provides higher-level thread functionality.

Let's now examine the working of the `threading` module.

The `threading` Module

The `threading` module exposes all the functions of the thread module and provides some additional functions:

`threading.activeCount().` Returns the number of thread objects that are active.

`threading.currentThread().` Returns the number of thread objects in the caller's thread control.

`threading.enumerate().` Returns a list of all thread objects that are currently active.

In addition to the functions, the `threading` module has the `Thread` class that implements threading.

The `Thread` Class

The `Thread` class encapsulates the functionality of the thread of execution. This class defines a number of methods that help create and manage threads. Table 13.1 lists the methods of the `Thread` class.

The `Thread` class is used to create threads. There are two ways of starting a thread after subclassing the `Thread` class. You can pass a callable function to the constructor of the subclass or override the `run()` method in the subclass. Only the `__init__()` method and the `run()` method of the `Thread` class should be overridden. After you create the thread of the object of the subclass, you can start the thread by calling the `start()` method of the thread. This invokes the `run()` method in a separate thread of control.

After the execution of the statements in a thread has started, it can be in any of the following states:

Alive and active. A thread enters alive and an active state when its activity starts.

Blocked. A thread enters a blocked state when the sleep function is called.

Dead. A thread can die when the run function terminates normally or when the run method is terminated abnormally due to an exception.

Table 13.1 Methods in the `Thread` Class

METHOD NAME	DESCRIPTION
`run()`	The `run()` method is the entry point for a thread.
`start()`	The `start()` method starts a thread by calling the `run` method.
`join([time])`	The `join()` waits for threads to terminate.
`isAlive()`	The `isAlive()` method checks whether a thread is still executing.
`getName()`	The `getName()` method returns the name of a thread.
`setName()`	The `setName()` method sets the name of a thread.

Having looked so far at the functions of the Thread class, let's examine the steps involved in creating a single-threaded application:

1. Extending the Thread class.
2. Defining the run() method.
3. Creating an object of the Thread class and calling the start() method.

Let's consider each of these steps separately.

Extending the Thread class. You can create a new class that is extended from the Thread class to implement the functionality that you require. The following code creates a class named MyThread.

```
import Threading
class MyThread(Threading.Thread):
```

Defining the run() method. You can override the run() method of the Thread class to meet your requirements. The following code sample prints the Hello World message 10 times after a time gap of 5 seconds.

```
def run(self):
     number =1
     while(number <= 10)
          print("Hello World")
          time.sleep(0.001)
```

Creating an object of the Thread class and calling the start() method. The start() method starts a thread by invoking the run() method of the Thread class.

```
NewThreadObject = MyThread()
NewThreadObject.start()
```

The main difference between instantiating the Thread class and calling the thread.start_new_thread() method is that the new thread object created by instantiating the Thread class need not start immediately, whereas the start_new_thread() method starts the thread right away. This can be a useful synchronization technique when you want all the thread objects to be created first and then later to start all the threads together.

Having looked at the different sections of the code, let's now look at the complete code.

```
import time
import Threading
class MyThread(Threading.Thread):
def run(self):
     number =1
     while(number <= 10)
          print("Hello World")
          time.sleep(0.001)
NewThreadObject = MyThread()
NewThreadObject.start()
```

Let's now consider converting this single-threaded application into a multithreaded application. All it requires is instantiating the `MyThread` class as many times as the number of threads you want to create and the `join()` method. The `threadobj.join()` method works as a synchronization mechanism. This method ensures that the program control does not exit the main loop until a thread (`threadobj`) terminates. This method is not important when the code is such that it will wait anyway for all the threads to complete execution before exiting out of the main loop. Here is the code in action.

```
from time import sleep, ctime, time
import threading
class MyThread(threading.Thread):
    def run(self):
        number =1
        while(number <= 5):
            print"Thread executing",ctime(time())
            sleep(1)
            number=number+1
threadarray = []
threadnumber = 1
while threadnumber <= 2:
    NewThreadObject = MyThread()
    NewThreadObject.start()
    threadarray.append(NewThreadObject)
    threadnumber=threadnumber+1
for mythread in threadarray:
     mythread.join()
print "End of My Code"
```

In the preceding code, we first create a subclass `MyThread` from the `Thread` class. The `run()` method of `MyThread` prints the current time five times after an interval of one second. Next, we create two thread objects by instantiating `MyThread`. The `start()` method is used to invoke the `run()` method of `MyThread`. Notice that a loop containing the `join()` method at the end of the code ensures that both threads are terminated before exiting from the code. The output of the preceding code will be this:

```
Thread executing Wed Jan 09 09:58:30 2002
Thread executing Wed Jan 09 09:58:30 2002
Thread executing Wed Jan 09 09:58:31 2002
Thread executing Wed Jan 09 09:58:31 2002
Thread executing Wed Jan 09 09:58:32 2002
Thread executing Wed Jan 09 09:58:32 2002
Thread executing Wed Jan 09 09:58:33 2002
Thread executing Wed Jan 09 09:58:33 2002
Thread executing Wed Jan 09 09:58:34 2002
Thread executing Wed Jan 09 09:58:34 2002
End of My Code
```

Let's add the functionality of displaying the thread number by overriding the constructor of the `Thread` class in `MyThread`.

```
from time import sleep, ctime, time
import threading
class MyThread(threading.Thread):
    def __init__(self,func,args):
        threading.Thread.__init__(self)
        self.func=func
        self.args=args
    def run(self):
        apply(self.func,self.args)
def func1(threadn,):
    number =1
    while(number <= 5):
        print"Thread no.",threadn,ctime(time())
        sleep(threadn)
        number=number+1

threadarray = []
threadnumber = 1
while threadnumber <= 2:
    NewThreadObject = MyThread(func1,(threadnumber,))
    NewThreadObject.start()
    threadarray.append(NewThreadObject)
    threadnumber=threadnumber+1
for mythread in threadarray:
     mythread.join()
print "End of My Code"
```

Notice that the constructor of `MyThread` can be used to pass any parameters to the `run()` method. In the preceding code, instead of processing threads in the `run()` method directly, we call another function, `func1()`. The parameters required by `func1()` are passed as a sequence along with the function's name while creating the instance of `MyThread`. All these arguments are initialized in the constructor of `MyThread`. The `run()` method invokes `func1()` and passes the sequence of arguments to `func1()`. The arguments passed have to be in the form of a sequence; therefore, a comma is specified after `threadnumber` while creating an instance of the `MyThread` class. Notice that `threadnumber` is the variable that contains the number of the thread that is currently executing.

```
The output of the preceding code will be:
Thread no. 1 Wed Jan 09 11:28:35 2002
Thread no. 2 Wed Jan 09 11:28:35 2002
Thread no. 1 Wed Jan 09 11:28:36 2002
Thread no. 2 Wed Jan 09 11:28:37 2002
Thread no. 1 Wed Jan 09 11:28:37 2002
Thread no. 1 Wed Jan 09 11:28:38 2002
Thread no. 2 Wed Jan 09 11:28:39 2002
Thread no. 1 Wed Jan 09 11:28:39 2002
Thread no. 2 Wed Jan 09 11:28:41 2002
Thread no. 2 Wed Jan 09 11:28:43 2002
End of My Code
```

Result

For creating an application for the scenario given earlier in this chapter, you need to create a server program and a client program. The server program will be running on a single computer whereas the client program can run on four computers at maximum. From the preceding discussion about `thread` and `threading` modules, we have identified that the `Thread` class is more effective for contemporary threading and synchronizing multiple threads. Therefore, the server program will incorporate the `Thread` class for creating four client threads. Let's write the code for the server and the client computers for the Techsity University scenario.

Write Code for the Server

The following is the code for creating a TCP server for the required application:

```
#server program
from socket import *
from time import sleep,time,ctime
import threading
ServerData=[]
Hostname = ''
PortNumber = 12345
Buffer = 500
ServerAddress = (Hostname, PortNumber)
#Create server socket
TCP_Server_Socket = socket(AF_INET, SOCK_STREAM)
TCP_Server_Socket.bind(ServerAddress)
TCP_Server_Socket.listen(4)
print 'Server is waiting for connection'
add=[]
class MyThread(threading.Thread):
    def run(self):
        while 1:
            TCP_Client_Socket, ClientAddress =\
                  TCP_Server_Socket.accept()
            print 'Server has accepted the connection request from ',\
                           ClientAddress
            if ServerData:
                for i in range(len(ServerData)):
                    TCP_Client_Socket.send(str(ServerData[i]))
                    sleep(0.01)
            else:
                    TCP_Client_Socket.send("Hi")
            print 'The Server is ready to receive data from \
                              the client'
            while 1:
                ClientData = TCP_Client_Socket.recv(Buffer)
                if not ClientData:
                    print 'The client has closed the connection'
                    break
```

```
                print 'The %s has sent this data string: %s'\
                              % (ClientAddress,ClientData)
                ClientData=ClientData+'~~'
                #Collect the data sent by all clients in ServerData
                ServerData.append(ClientData)
                #send the data collected in ServerData
                for i in range(len(ServerData)):
                    TCP_Client_Socket.send(str(ServerData[i]))
                    sleep(0.01)
                print 'The Server is ready to receive more data from\
                                    the client'

            TCP_Client_Socket.close()
            break
        TCP_Server_Socket.close()

ch=0
while ch<=3:
    #Create four threads for four clients
    NewThreadObject = MyThread()
    NewThreadObject.start()
    threadarray.append(NewThreadObject)
    ch=ch+1
```

Write the Code for the Client

The following is the code for creating a TCP client for the required application:

```
# Client program
from socket import *
from time import sleep
Hostname = 'localhost'
PortNumber = 12345
Buffer = 500
#Establish connection with the server
ServerAddress = (Hostname, PortNumber)
TCP_Client_Socket = socket(AF_INET, SOCK_STREAM)
TCP_Client_Socket.connect(ServerAddress)

while 1:
    print 'The client is connected to the server'
    ServerData = TCP_Client_Socket.recv(Buffer)
    if not ServerData:
        print 'The server has sent nothing'
        break
    else:
        #process data received
        ServerStr=str(ServerData)
        if ServerStr.find('~~')!=-1:
            ServerList=ServerStr.split('~~')
```

```
            for i in range(len(ServerList)):
                print ServerList[i]
        else:
            print ServerStr
    DataStr = raw_input('Enter data to broadcast: ')
    if not DataStr:
        print 'The client has entered nothing; hence the connection\
                       to the server is closed'
        break
    #send data
    TCP_Client_Socket.send(DataStr)
    sleep(0.1)
    #receive data from server

TCP_Client_Socket.close()
```

Execute the Code Created for the Server

To implement or view the output of the server program, perform the following steps on the server computer:

1. Write the server code in a text editor and save it as **MultiThrServer.py**.
2. At the shell prompt, type:

   ```
   $ python MultiThrServer.py
   ```

The server starts as shown in Figure 13.1.

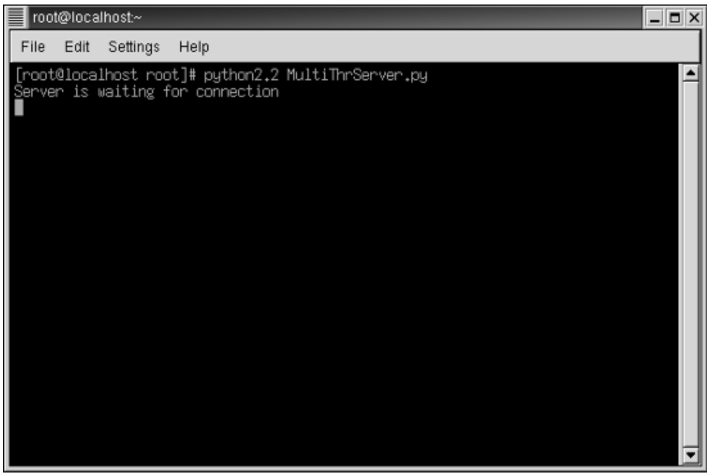


Figure 13.1 The screen after starting the server.

Execute the Code Created for the Client

To implement or view the output of the client program, perform the following steps on the client computer:

1. Write the client code in a text editor and save it as **MultiThrClient.py**.
2. At the shell prompt, type:

   ```
   $ python MultiThrServer.py
   ```

3. At the prompt `Enter data:`, enter:

   ```
   How do I start a comment in Python?
   ```

   Figure 13.2 shows a client connected and sending data to the server.
4. The server sends the data written to the file back to the client.
5. Open another terminal window, and start the client program again. Notice that the message sent by the previous client appears here.
6. At the prompt `Enter data:`, enter:

   ```
   In python, comments begin with a pound (#) sign.
   ```

   Figure 13.3 shows another client connected and sending data to the server.

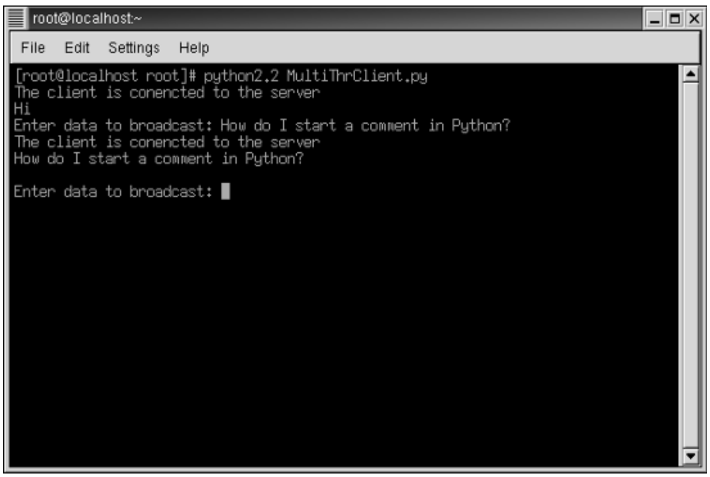


Figure 13.2 The client sending data to the server.

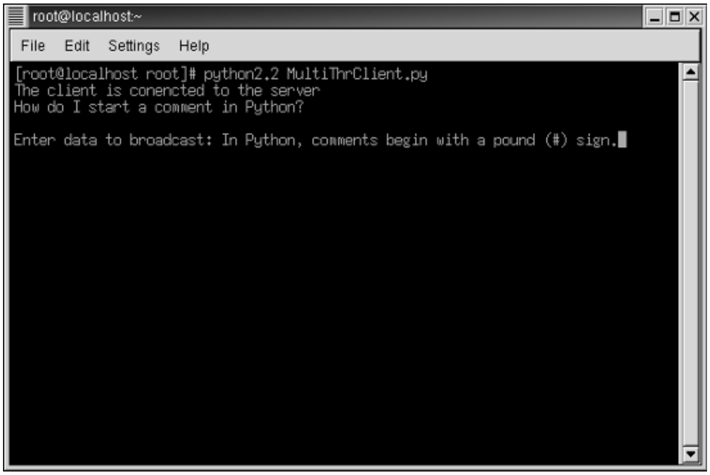


Figure 13.3 The second client sending data to the server.

7. Once again, the server sends the data back to the second client, as shown in Figure 13.4.
8. Open another terminal window and start the client program again. Notice that the messages sent by both the previous clients appear. (Refer to Figure 13.5.)

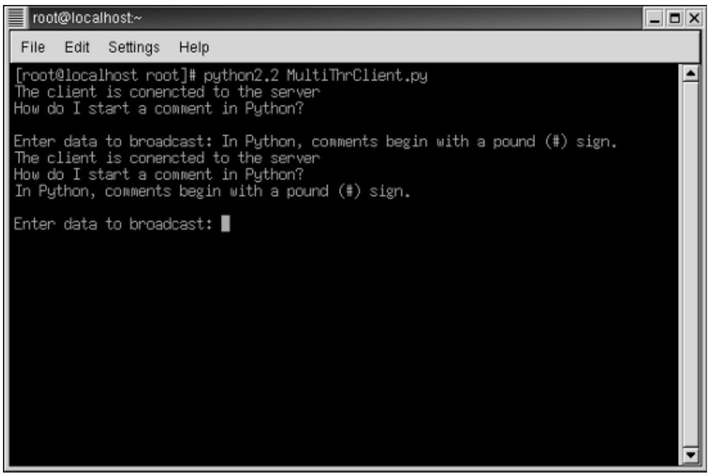


Figure 13.4 The server sending data to the sending client.

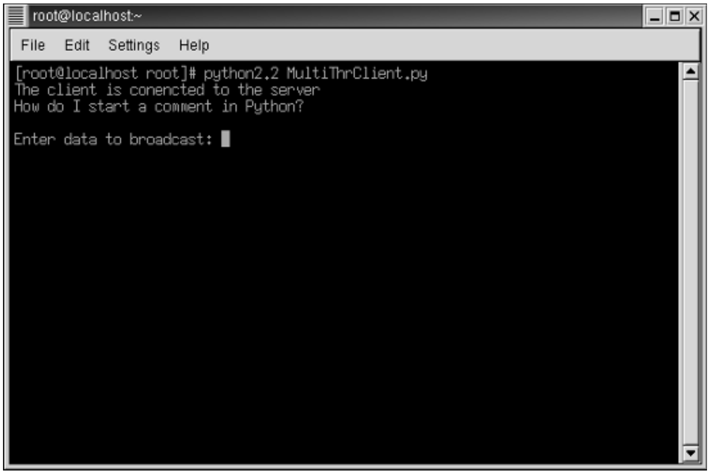


Figure 13.5 The third client receiving data from the server.

9. To exit from all the client programs, at the prompt `Enter data:`, press the Enter key.

NOTE **Regularly observe the output of all programs to understand the communication between the server and clients. The server program will not end on its own because it is in an infinite loop. You will have to close the terminal window in which the server program is running to end it.**

Summary

In this chapter, you learned the following:

- A *thread* is the smallest unit of code that can be executed. Any program that has more than one thread is called a multithreaded program.
- A *process* is an executing instance of a program.
- A single-threaded application has only one thread. In a single-threaded application, user input and any processing that does not require user input are handled by the same thread.
- Python provides two modules to support multithreaded programming:
  - `thread`
  - `threading`

- The `thread` module provides the basic thread and locking features and is appropriate for lower-level thread access, whereas the `threading` module provides higher-level thread functionality.
- The most commonly used thread functions available in the `thread` module are as follows:
  - `thread.start_new_thread(func, args,[,kwargs])`
  - `thread.exit()`
  - `thread.get_thread()`
  - `thread.allocate_lock()`
- The following functions are exposed to a lock object:
  - `lock_obj.acquire([waitflag])`
  - `lock_obj.release()`
  - `lock_obj.locked()`
- The `threading` module exposes all the functions of the thread module and provides some additional functions. These functions are as follows:
  - `threading.activeCount()`
  - `threading.currentThread()`
  - `threading.enumerate()`
- In addition to the functions, the `threading` module has the `Thread` class that implements threading. The methods provided by the `Thread` class are as follows:
  - `run()`
  - `start()`
  - `join([time])`
  - `isAlive()`
  - `getName()`
  - `setName()`
- The steps involved in creating a single-threaded application are these:
  1. Extending the `Thread` class
  2. Defining the `run()` method
  3. Creating an object of the `Thread` class and calling the `start()` method
- Creating a multithreaded application also requires the same steps except that you have to specify a join method for each thread object to ensure that the program does not exit the main loop as all threads terminate.

CHAPTER

14

Advanced Web Programming

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Create Web servers by using:**
  - **The** `SocketServer` **module**
  - **The** `BaseHTTPServer` **module**
  - **The** `SimpleHTTPServer` **module**
  - **The** `CGIHTTPServer` **module**
- **Access URLs in Python by using:**
  - **The** `urlparse` **module**
  - **The** `urllib` **module**
- **Upload files across an HTTP connection**
- **Use cookies for data persistence on the client side**

Getting Started

Web programming and network programming were introduced in Chapter 10, "CGI Programming," and Chapter 12, "Network Programming." You are now familiar with the way data is handled over networks and how it can be transferred from a client to a server or vice versa using Python. This chapter takes you further and discusses advanced Web programming concepts. To start, this chapter discusses how to create your own Web server. Next, it talks about how you can work with URLs by using Python. Finally, this chapter explains advanced CGI to generate dynamic Web pages using cookies and uploading files across an HTTP connection.

Creating Web Servers

You already know that CGI request processing involves Web servers and clients. Usually, we use browsers, such as Netscape Navigator and Internet Explorer, as Web clients. The most popular Web servers are IIS, Apache, and Netscape. You can use Python for creating a Web server and Web clients. Some of the modules available in Python for building Web servers are `SocketServer`, `BaseHTTPServer`, `SimpleHTTPServer`, and `CGIHTTPServer`.

The `SocketServer` Module

The `SocketServer` module is used for creating general IP servers. It provides the necessary framework for network servers and simplifies the job of writing them. While using this module, you do not require the `socket` module to implement servers. This module implements servers by using four classes: `TCPServer`, `UDPServer`, `UnixStreamServer`, and `UnixDatagramServer`. These classes provide interfaces to the most commonly used protocols, and they handle requests in a synchronous manner. The `UnixStreamServer` and `UnixDatagramServer` classes use Unix domain sockets and are not meant for non-Unix platforms. The `TCPServer and UDPServer` classes implement a server and support the TCP protocol and the UDP protocol, respectively.

In addition, it also provides `StreamRequestHandler` and `DatagramRequestHandler` classes to handle requests. While using the `SocketServer` module, you can handle requests as separate threads.

The following steps explain the creation of a Web server by using the `SocketServer` module:

1. Create a request handler class by subclassing the `StreamRequestHandler` class or the `DatagramRequestHandler` class and overriding its `handle()` method. The `handle()` method processes incoming requests.
2. Instantiate one of the server classes by passing the address of the server and the instance of the request handler class.
3. Finally, call the `handle_request()` method to process the request of a client or the `serve_forever()` method of the server object to process the requests of many clients.

TCPServer, UDPServer, UnixStreamServer, and UnixDatagramServer classes require two parameters. The first parameter is a 2-tuple comprising the host name and the port of the server address; the second parameter is the request handler class, which is an instance of the BaseRequestHandler class. All of the classes discussed have their own instances of class variables; however, all of them implement the following methods and attributes:

fileno(). This method returns an integer file descriptor for the socket of the Web server that you create.

handle_request(). This method processes a single request at a time and invokes the handle() method of the handler class.

serve_forever(). This method handles an infinite number of requests by calling handle_request() inside an infinite loop.

address_family. This is the family of protocols of the server socket, socket.AF_INET and socket.AF_UNIX.

RequestHandlerClass. This is the class that handles all the requests through the handle() method; an instance of this class is created for each request.

server_address. This is the address containing the IP address and an integer port number of the socket on which the server is listening.

socket. This is the socket object on which the server listens for approaching requests.

The following code, **SocServer.py**, creates a Web server by using the SocketServer module. A TCP client or a UDP client can be used to connect to this server.

```
import SocketServer
from time import sleep
port=8888
class myR(SocketServer.StreamRequestHandler):
    def handle(self):
        print "connection from",self.client_address
        try:
            self.wfile.write("SocketServer works!")
        except IOError:
            print "Connection from the client ",\
                    self.client_address," closed"

while 1:
    srvsocket=SocketServer.TCPServer(("",port),myR)
print "the socket is listening to the port", port
srvsocket.serve_forever()
```

When you start the server and connect it with two clients, it displays the following:

```
the socket is listening to the port 8888
connection from ('127.0.0.1', 34181)
connection from ('127.0.0.1', 34182)
```

The serve_forever() method ensures that multiple clients can access the server.

The `BaseHTTPServer` Module

The BaseHTTPServer module provides the infrastructure required for creating HTTP servers (Web servers). To implement these servers, the BaseHTTPServer module defines two classes, HTTPServer and BaseHTTPRequestHandler.

The HTTPServer class is the subclass of TCPServer, which belongs to the ServerSocket module. You use this class to create Web sockets. The HTTPServer class also listens to Web sockets and directs requests to the concerned handler. The BaseHTTPRequestHandler class is used to handle HTTP requests. You need to create a subclass of the BaseHTTPRequestHandler class to handle each request method. The BaseHTTPRequestHandler class defines various instance variables, class variables, and methods that can be used by its subclasses. The following methods are exposed by the BaseHTTPRequestHandler class:

handle(). This method handles a request by calling the do_GET() method when the base server receives a GET request and by calling the do_POST() method when the base server receives a POST request.

send_error[error code[,error message]]. This method sends an error to the client along with the error code that is the HTTP error code and the message that is the optional text you can specify.

send_response[response code[,response message]]. This method sends a response header. You can specify a more specific response message also. For example, the response code 200 returns an "OK" status.

send_header[keyword,value]. This method sends the MIME header to the output stream where a keyword specifies the header keyword and a value specifies the header value.

end_headers(). This method indicates the end of MIME headers.

The following attributes are also exposed by the BaseHTTPRequestHandler class:

client_address. Returns a tuple containing the host and port number referring to the address of the client.

command. Returns the request type, such as GET, POST, etc.

path. Returns the request path.

request_version. Returns the version string from a request, such as HTTP/1.0.

headers. Contains the message headers in the HTTP request.

rfile. Contains the input stream used to read the data received from the client.

wfile. Contains the input stream used for sending a response to the client.

The following code, **Mywebserver.py**, illustrates a Web server created using the BaseHTTPServer module:

```
#!/usr/bin/python2.2
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
dynhtml="""
<HTML><HEAD><TITLE>My Home Page</TITLE></HEAD>
```

```
<BR><BR><BR><BR><BR><BR><HR>
<BODY><CENTER><H1><U>Hello client!</U></H1>
<H2>You are connected to Mywebserver</H2><HR></BODY>
</HTML>"""
nf="File not found"
class req_handler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path=="/":
            self.send_response(200)
            self.send_header('Content-type','text/html')
            self.end_headers()
            self.wfile.write(dynhtml)
        else:
            self.send_error(404,nf)
try:
     server=HTTPServer(('',8000),req_handler)
     print 'Welcome to the Mywebserver...'
     print 'Press ^C once or twice to quit'
     server.serve_forever()
except KeyboardInterrupt:
        print '^C pressed, shutting down server'
        server.socket.close()
```

When you execute the preceding code, a Web server starts. When a client tries to access the server, an OK status is returned with the response code 200 and the default page of the server is displayed, as shown in Figure 14.1.

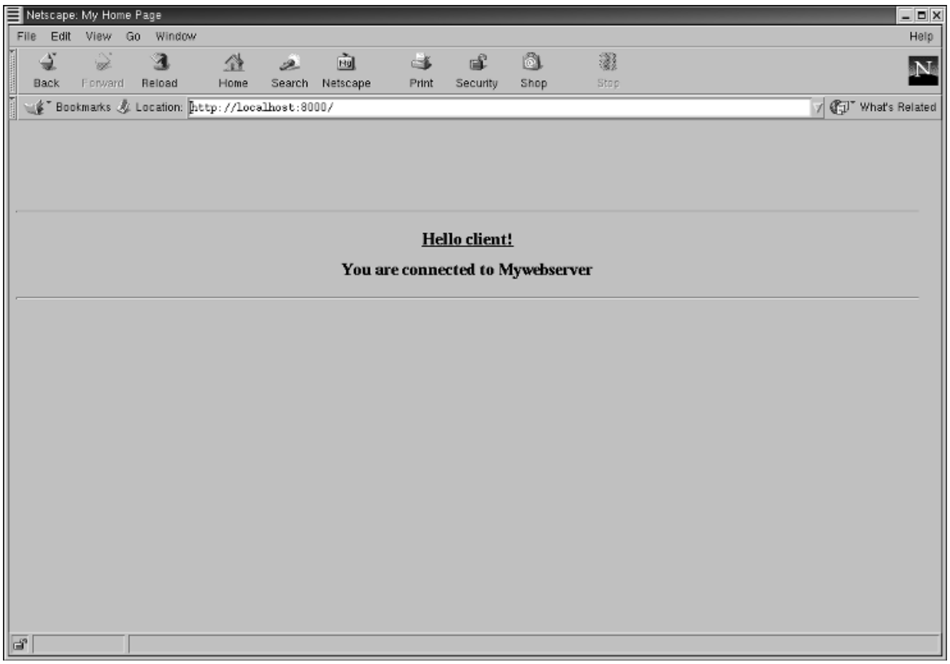


Figure 14.1 The default page displayed when the Web server is accessed.

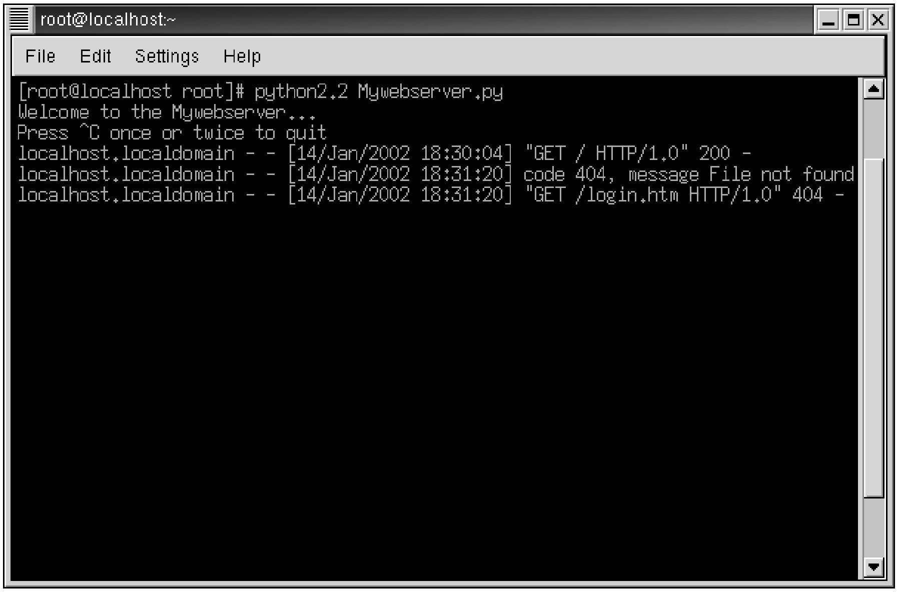


Figure 14.2 Accessing the Web server.

This is a simple program to create a Web server, and it cannot access files on the server. If the client tries to access a file, the "file not found" message code is displayed with the error code 404. The server displays loggable output as shown in Figure 14.2.

NOTE Standard Web servers run on port 80. Other Web servers that you create do not have access to this port. Therefore, if you are accessing a Web server from a client browser, do not forget to specify the server's port number along with its host name, which is the name of the computer on which the server resides.

The `SimpleHTTPServer` Module

The `SimpleHTTPServer` module is used for creating simple Web servers. It defines a class, `SimpleHTTPRequestHandler`, for handling requests to serve only base directory files. This module is compatible with the `BaseHTTPServer` module. Therefore, the methods and attributes exposed by the `SimpleHTTPServer` module are similar to the `BaseHTTPServer` module; however, this module is suitable for implementing GET and HEAD requests.

The following code, **Mysimplewebserver.py**, illustrates a Web server created using the `SimpleHTTPServer` module:

```
#!/usr/bin/python2.2
from os import curdir, sep
from BaseHTTPServer import HTTPServer
```

```
from SimpleHTTPServer import SimpleHTTPRequestHandler
class req_handler(SimpleHTTPRequestHandler):
    def do_GET(self):
        try:
            f = open(curdir + sep + self.path)
            self.send_response(200)
            self.send_header('Content-type','text/html')
            self.end_headers()
            self.wfile.write(f.read())
            f.close()
        except IOError:
            self.send_error(404,'File Not Found: %s' % self.path)
try:
     server=HTTPServer(('',8000),req_handler)
     print 'Welcome to the My simple web server...'
     print 'Press ^C once or twice to quit'
     server.serve_forever()
except KeyboardInterrupt:
        print '^C pressed, shutting down server'
        server.socket.close()
```

When you execute the preceding code, a Web server starts. When a client browser tries to open a Web page, the Web page is displayed, as shown in Figure 14.3.

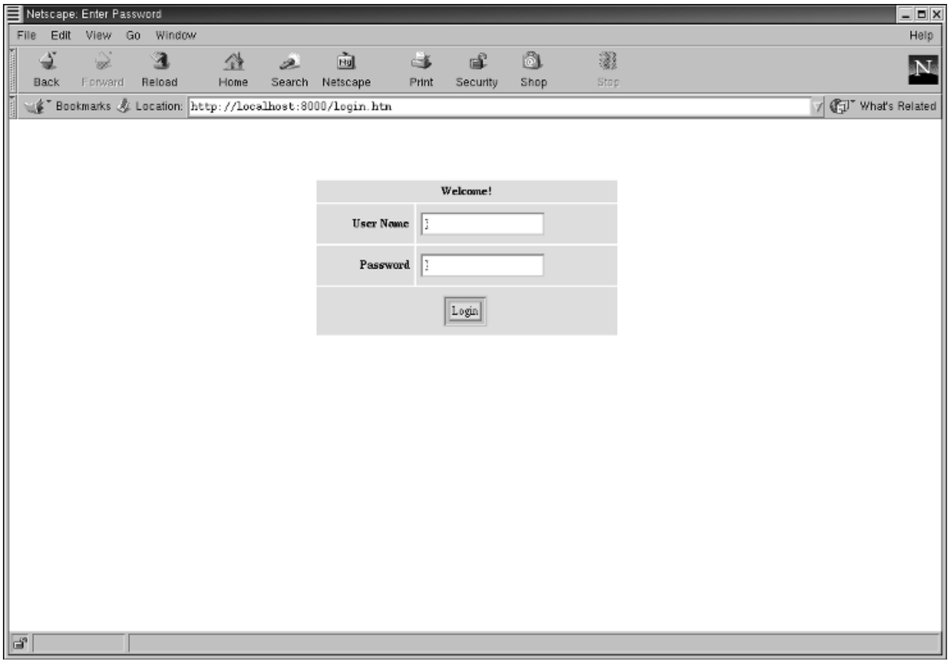


Figure 14.3 The page displayed when the Web server is accessed.

This code can access files using the server and send the response code 200 if the file is found. If the client tries to access a file that does not exist, the "file not found" message code is displayed with the error code 404. The server displays output that can be logged, as shown in Figure 14.4.

The `CGIHTTPServer` Module

The `CGIHTTPServer` module is used for creating Web servers that support CGI. It defines a class, `CGIHTTPRequestHandler`, for handling requests or the output of CGI scripts. This module is compatible with the `BaseHTTPServer` module. The `CGIHTTPRequestHandler` class also inherits its behavior from `SimpleHTTPRequestHandler` with an added functionality of handling CGI scripts. To handle CGI requests, the `CGIHTTPRequestHandler` class implements the `cgi_directories` attribute. This attribute contains a list of directories in which CGI scripts can be stored.

Your Web surfing experience must have made you familiar with URLs. (URLs were introduced in Chapter 10.) Let's discuss functions that can be used to work with URLs.

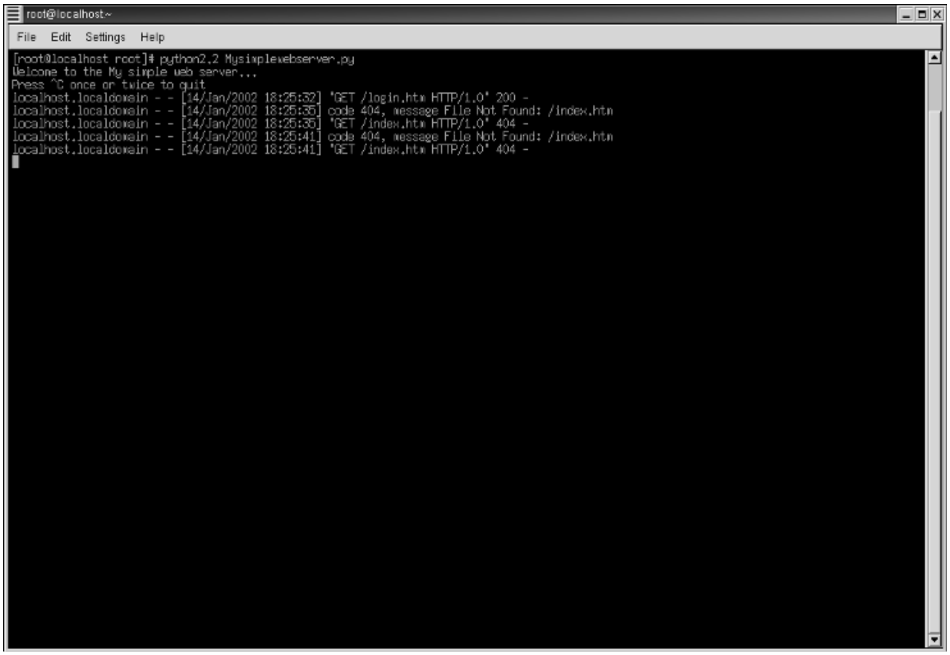


Figure 14.4 The Web page opened by the Web server.

Accessing URLs

While accessing a simple Web page or executing a CGI script, you come across URLs. Many times, you may need to process the URLs in your scripts. In Chapter 10, we said that a typical URL consists of a protocol, a host, a domain name, a top-level domain name, and a path. It may contain a few other components as well. For example, you must have noticed that when a Web page sends a GET request to a Web server, the URL passed contains other components as well. You usually see URLs of the following format on the Web:

```
protocol://server_loc/path:params?query#frag
```

Let's understand each of these components:

protocol refers to the type of protocol to be used.

server_loc refers to the location of the server where the resource to be accessed or user information is stored. User information can be stored in this component of URL in the form

```
user:password@host:port
```

path refers to the path of a file or a CGI application separated by slashes.

params refers to optional parameters.

query refers to key-value pairs separated by ampersands (&).

frag refers to fragment identifier within a document.

Python provides `urllib` and `urlparse` modules to process URLs. Let's discuss the functions provided by these modules.

The `urlparse` Module

The `urlparse` module provides the functionality of manipulating URL strings. It contains functions to break a URL into tuples, as well as to combine them back to form the original URL. These functions are `urlparse()`, `urlunparse()`, and `urljoin()`.

The `urlparse.urlparse()` Function

The `urlparse()` function breaks any URL into a 6-tuple containing the elements described previously, which are `protocol`, `server_loc`, `path`, `params`, `query`, and `frag`. The syntax of the `urlparse()` function is:

```
urlparse(url[,def_prot_scheme[,allow_frags]])
```

For example,

```
>>> import urlparse
>>> urlparse.urlparse('http://www.python.org/doc/lib/lib.html')
('http', 'www.python.org', '/doc/lib/lib.html', '', '', '')
```

The `urlparse.urlunparse()` Function

The `urlunparse()` function combines the components of a URL returned by the `urlparse()` function back to form the original URL. In other words, the tuple containing six elements (`protocol`, `server_loc`, `path`, `params`, `query`, and `frag`) returned by the `urlparse()` function is joined to form a URL by the `urlunparse()` function. The syntax of the `urlunparse()` function is this:

```
urlunparse(urltuple)
```

For example,

```
>>> urltup=('http', 'www.python.org', '/doc/lib/lib.html')
>>> urlparse.urlunparse(urltup)
'http://www.python.org/doc/lib/lib.html'
```

The `urlparse.urljoin()` Function

The `urljoin()` function combines two URLs after excluding the filename from the first URL. To understand this, let's first look at the syntax of `urljoin()`:

```
urlparse.urljoin(baseurl, url [,allowfrag])
```

The `urljoin()` function joins `baseurl` with `url` after removing the filename, if provided, from `baseurl`. For example,

```
>>> urlparse.urljoin("http://www.python.org/doc/Newbies.html",\
                        "current/tut/tut.html")
'http://www.python.org/doc/current/tut/tut.html'
```

The `urllib` Module

The `urllib` module provides the functionality of retrieving data from the Web by using the given URLs. It also provides functions for encoding and decoding strings so that they can be suitably used as parts of URL strings.

The `urllip.urlopen()` Function

The `urlopen()` function retrieves a Web page and returns a file-like object that can be manipulated the way any other file object can be manipulated. The syntax of the `urlopen()` function is this:

```
urlopen(url,[,encoded_data])
```

The `urlopen()` function opens the URL `url`. For HTTP GET requests, a query string should be provided as part of `url`. You will learn about encoding data in the next section, the `urlencode()` function. For POST requests, because data is to be kept a secret, it is not passed as part of `url`. Instead, encoded data is passed in

`encoded_data`. Let's write a program, **urlopenmod.py**, to illustrate the retrieval of a Web page from www.python.org and copy the Web page to another local file.

```
import urllib
copyfile=open("copypage.html","wb")
f=urllib.urlopen("http://www.python.org/")
data=f.read(500)
while data:
    copyfile.write(data)
    data=f.read(500)
copyfile.close()
f.close()
```

The preceding code opens the home page of the official Web site of Python and transfers it to another HTML file. The code transfers only 500 bytes at a time. The `urllib.urlopen ("http://www.python.org/")` command creates a stream object. All the usual file object functions can be used with this object. Figure 14.5 shows the **copypage.html** file when opened in a browser.

This stream object provides two additional attributes, `url` and `headers`. `url` contains the URL of the page that you are opening, and `headers` contains a dictionary that contains page headers. Let's change the preceding code to **urlopenmod1.py** to illustrate the usage of these attributes.

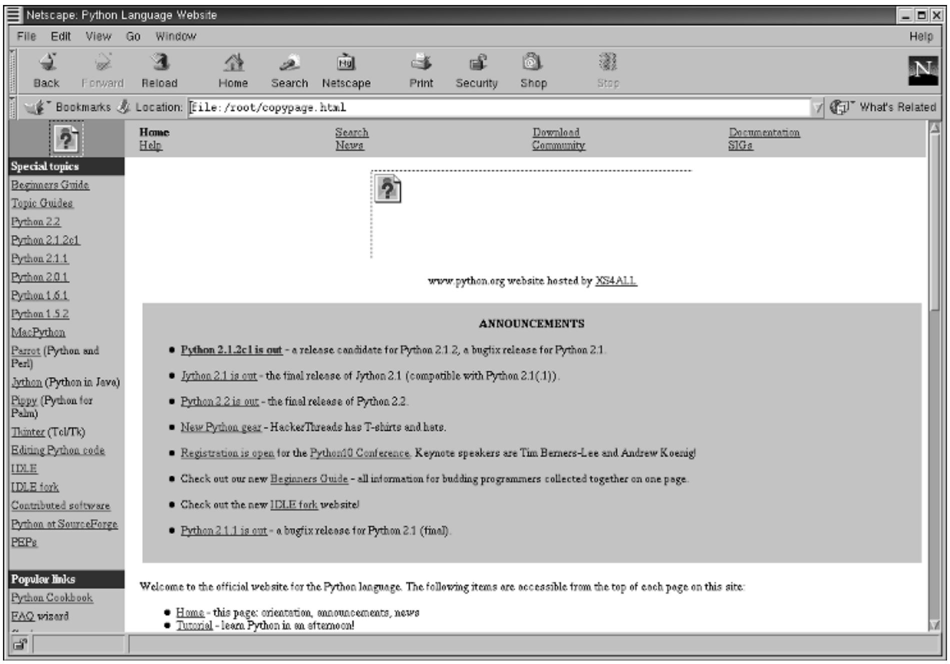


Figure 14.5 The copypage.html file.

```
import urllib
copyfile=open("copypage.html","wb")
f=urllib.urlopen("http://www.python.org/")
data=f.read(500)
while data:
    copyfile.write(data)
    data=f.read(500)
print "URL:", f.url
for key, value in f.headers.items():
    print key,"=",value
copyfile.close()
f.close()
```

The preceding code generates the following output at the Python prompt:

```
URL: http://www.python.org/
content-length = 13432
keep-alive = timeout=15, max=100
server = Apache/1.3.20 (Unix)
last-modified = Fri, 11 Jan 2002 03:49:53 GMT
connection = close
etag = "5a7511-3478-3c3e60e1"
date = Fri, 11 Jan 2002 06:46:32 GMT
content-type = text/html
accept-ranges = bytes
```

The `urllib.urlretrieve()` Function

The `urlretrieve()` function does all the work that is performed by **urlopenmod.py**. In other words, the `urlretrieve()` function opens a Web page that is specified in the network path and copies the Web page to a local file. The syntax of this function is this:

```
urlretrieve(url[,filename[,downloadstatushook]]
```

For example,

```
urllib.urlretrieve("http://www.python.org/","copypage.html")
```

This command copies the entire home page of `www.python.org` to `copypage.html`. The optional argument, `downloadstatushook`, is the name of the function that is executed after each block of data is copied to a local file.

The `urllib.quote()` and `urllib.quote_plus()` Functions

The `quote()` function converts a string to its encoded version by replacing each special character with its `%xx` escape code. `%xx` code is the hexadecimal representation of a character. The syntax of the `quote()` function is this:

```
quote(string[,safe])
```

For example,

```
>>>urllib.quote('http://search.python.org/query.html?qt=CGI&\
                        col=ftp&col=python')
'http%3A//search.python.org/query.html%3Fqt%3DCGI%26col%3Dftp%26col%3Dpy
thon'
```

In the `safe` string, you can specify the set characters that you do not want to be converted. However, certain characters are never converted: commas, underscores, dashes, periods, and alphanumeric characters.

The `quote_plus()` function also works like the `quote()` function except that the `quote_plus ()` function replaces spaces by plus (+) signs.

```
>>>urllib.quote_plus('http://search.python.org/query.html?\
                        qt=CGI COM&col=ftp&col=python')
'http%3A//search.python.org/query.html%3Fqt%3DCGI+COM%26col%3Dftp%26col%
3Dpython'
```

The `urllib.unquote()` and `urllib.unquote_plus()` Functions

The `unquote()` function converts an encoded string back to the original string. Therefore, when you supply an encoded string containing `%xx` codes, the codes are converted to their ASCII equivalents. The syntax of the `unquote()` function is this:

```
unquote(string)
```

For example,

```
>>> urllib.unquote('http%3A//search.python.org/query.html%3Fqt\
                                %3DCGI%26col%3Dftp')
'http://search.python.org/query.html?qt=CGI&col=ftp'
```

The `unquote_plus()` function is similar to the `unquote()` function, and it converts plus signs to spaces. For example,

```
>>> urllib.unquote_plus('http%3A//search.python.org/query.html%3Fqt\
                     %3DCGI+COM%26col%3Dftp%26col%3Dpython')
'http://search.python.org/query.html?qt=CGI COM&col=ftp&col=python'
```

The `urllib.urlencode()` Function

The `urlencode()` function takes a dictionary and converts it into a URL-encoded string that can be included as a part of the query in the CGI request string. The key-value pairs are first encoded in the "key=value" format with each key-value pair separated by an ampersand (&). The syntax of the `urlencode()` function is this:

```
urllib.urlencode(dict)
```

For example,

```
>>> dict1={'name':'Laura', 'studid':'S001'}
>>> urllib.urlencode(dict1)
'name=Laura&studid=S001'
```

Having considered the `urlparse` and `urllib` modules to work with URLs, we will now consider how to develop advanced CGI applications.

Creating Advanced CGI Applications

Problem Statement

Besides offering Web-based training courses, Techsity University also provides instructor-led training courses. To enhance and test the learning of students, the trainers in the university have unanimously agreed to send a weekend assignment to students every week. Catherine, a Web site designer, is assigned the task of preparing a form on the Web site. A student should be able to enter personal details in the form and upload an assignment to the university's Web server. The details in the form should include student ID, student name, course ID, and assignment number. After a student clicks the Submit button on the form, these details and the file uploaded should appear on the next screen in the browser. The student should also be able to go back to the form and view the details previously entered.

Task List

- **Identify the elements of the Web page for entering assignment details and uploading the file.**
- **Identify the methodology for uploading a file.**
- **Identify the methodology for storing user information.**
- **Write the code for the CGI script.**
- **Execute the CGI script.**

Identify the Elements of the Web Page for Entering Assignment Details and Uploading the File

A form has to be designed as a user interface to gather the required information from a student. See Table 14.1.

Table 14.1 Elements of the User Interface for Entering Assignment Details

DETAIL	TYPE
Student Name	Text box
Student ID	Text box
Course ID	Text box
Assignment Number	Text box
File Name/Path	A box with a button to browse to the path of the file to be uploaded

Identify the Methodology for Uploading the File

A file can be uploaded using the file input type:

```
<input type="file" name="File_Upload" value="/root/abc.txt">
```

This directive adds an empty text box with a button on its side, which allows you to browse to the path of the file you want to upload. In most browsers, this button reads Browse. However, some browsers might label it with ellipses (...).

To upload a file to the Web server through a form, you should specify the form encoding as `multipart/formdata`. The default form encoding is `application/x-www-form-urlencoded`. Therefore, you do not need to specify this encoding with the FORM tag. For multipart forms, encoding should be specified as follows:

```
<form method="post" action="/cgi-bin/mycgi.py" enctype="multipart/
form-data">
```

Let's consider a CGI script, **uploadfile.py**, which uploads a file to a Web server.

```
#!/usr/local/bin/python2.2
from cgi import FieldStorage
header="Content-type:text/html\n\n"
dynhtml="""<html>
<head>
<title>File upload</title>
</head>
<body>
<form action="/cgi-bin/uploadfile.py" method="POST"
enctype="multipart/form-data">
```

```
<input type="file" name="file_name" size="50">
<input type="submit">
</form>
</body>
</html>"""
shtml = '''<HTML><HEAD><TITLE>
</TITLE></HEAD>
<BODY>
<H3>Contents: %s</H3>
<PRE>%s
</PRE>
</BODY></HTML>'''
form=FieldStorage()
if not form:
     print header+dynhtml
elif form.has_key("file_name"):
     fileupload=form["file_name"]
     data=' '
     if fileupload.file:
          count=0
          while 1:
               line=fileupload.file.readline()
               data=data+line
               if not line:
                    break
               count=count+1
          print header+shtml % (fileupload.filename,data)
else:
     pass
```

The preceding code is a simple code that sends a file across an HTTP connection using an HTML form. Notice that the file input type is used to submit multipart form data to the Web server. The `file` attribute of the uploaded file is used to read data from the uploaded file, and the `filename` attribute of the uploaded file is used to return the name of the file. When the CGI script is accessed, the screen shown in Figure 14.6 is displayed.

When you use the Browse button on the form to navigate to the file you want to upload and then click the Submit query button, the screen shown in Figure 14.7 is displayed in the browser.

Identify the Methodology for Storing User Information

We will use cookies as the methodology to store user information. Cookies are pieces of information stored by a Web server on a client computer and managed by a browser. When a client sends a request to a server for a site visited previously, the server accesses these cookies to retrieve site-related information. In the case of a request from

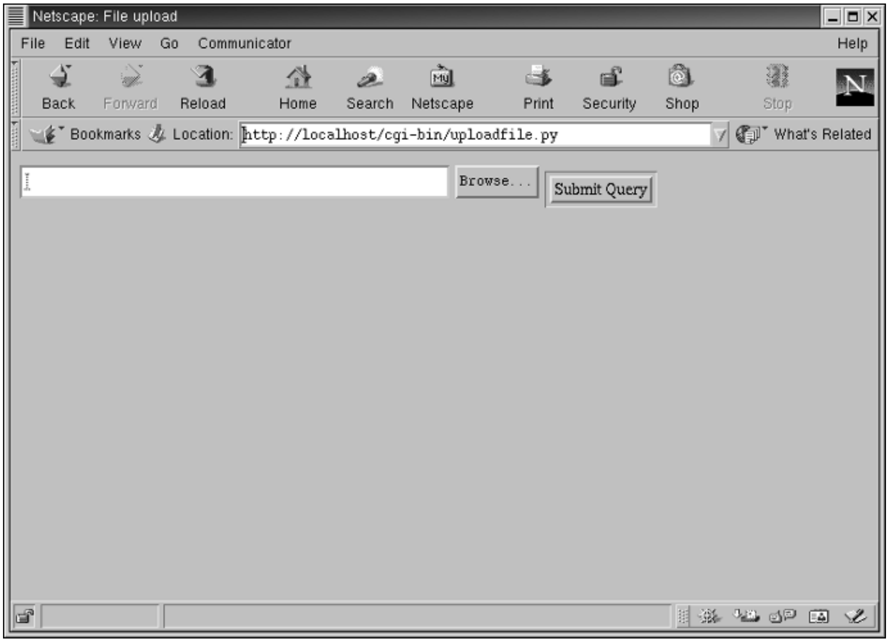


Figure 14.6 A form showing file input type.

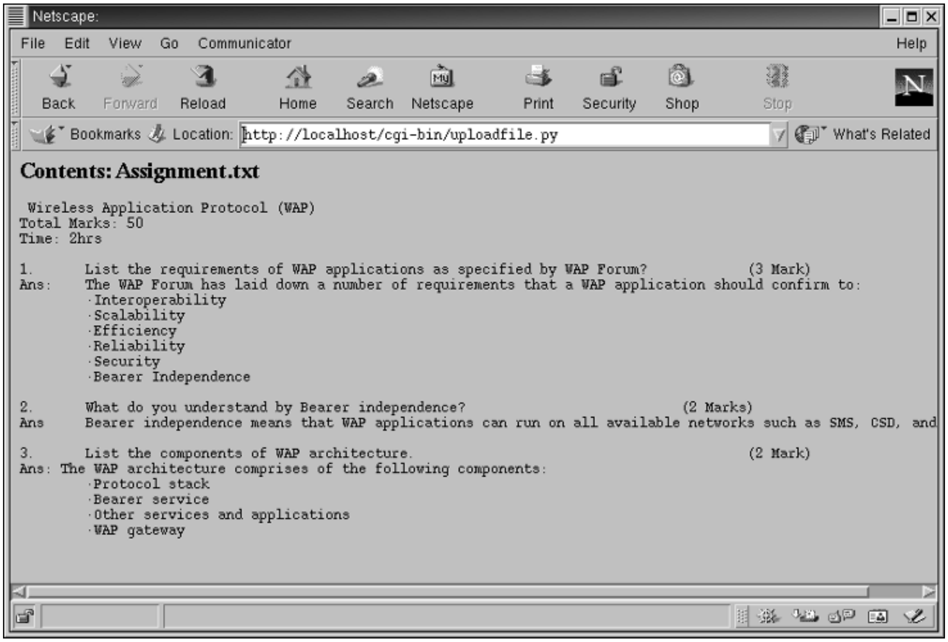


Figure 14.7 A sample Web page showing the contents of the uploaded file.

a CGI script, the same methodology is applied to retrieve information regarding repeated requests for the same page. As a result, cookies can be used in the following cases:

User identification. Cookies provide a link for identifying a user on subsequent visits to a Web site. Such a methodology can be explained using the example of an online store. After adding an item to the shopping cart, the HTTP connection is closed. How is the server able to identify the user during subsequent item selections? The Web server stores a cookie on a client computer by requesting the client computer to create a cookie. The cookie is sent back to the Web server in the form of HTTP headers when a client sends a request to the server.

Username and password specifications. Cookies can be used to store user identification. Instead of specifying the username and password during subsequent visits to a Web site, cookies can be used to store the username and the password. After a user registers on the Web site, a cookie is created with a unique ID that is associated with a specific user. When the user subsequently visits the site, the user's ID is used to identify the registered user without requiring username and password specifications. Such a methodology is used only for low-security sites.

Web page customizations. Cookies can store user-specified formats that are used to change the appearance of Web pages according to users' preferences. The changed format of a page is retained and can be retrieved during subsequent visits to the site.

Each cookie contains several bits of information, such as a variable name, a value, an expiration date, and a path. The variable name and the value are stored in a cookie in the form of key-value pairs separated by equal (=) signs and delimited by semicolons (;). The number of cookies and amount of information that can be stored on a computer can be limited to save hard disk space. The values must be shorter than 2KB, and the size of all cookies from one site must be less than 20KB total. The expiration date contains the time interval for retaining a cookie on a computer. The path in the cookie determines the locations on the site for the validity period of a cookie. For example, if your application is stored at the URL `/cgi-bin/app/myapp.py` and it sets a cookie whose path is `"/"`, then that cookie will be sent when you visit any URL on the whole site.

In Python, the `Cookie` module handles cookies. Let's examine the use of the `Cookie` module.

The `Cookie` Module

The `Cookie` module has many classes that handle the creation of cookie objects. The base class for the creation of all cookie objects is `Cookie`. A cookie object can be created using the `Cookie` class as follows:

```
>>> import Cookie
>>> cookie=Cookie.Cookie()
```

A cookie object contains key-value pairs and behaves like a dictionary. Therefore, a value can be assigned and extracted like a dictionary. This cookie object supports all the cookie attributes defined by RFC 2109. For example,

```
>>> cookie['studname']='Laura Jones'
>>> cookie['studid']=200
>>> cookie['studid'].value
200
```

When assigning value to cookies, nonstring objects are converted to string objects. A cookie can be displayed as follows:

```
>>> print cookie
Set-Cookie: studname="Laura Jones";
Set-Cookie: studid="I200\012.";
```

Let's now write the code for calculating the hit count for a Web page. The following CGI script, **hitcount.py**, calculates the number of times a client visits a Web page, assigns a random ID to the client, and returns the hit count and the ID to the client.

```
#!/usr/bin/python2.2
import Cookie
import cgi
import os
from random import randint
dynhtml='''<HTML><HEAD><TITLE>
Hit Count</TITLE></HEAD>
<HR><CENTER><BODY><H2>You have visited this page %s time(s)</H2>
<p><H3>Your visitor ID is: <b>%s</b></p></H3><CENTER>
<HR>
</BODY></HTML>'''
def getCookie(initialvalues={}):
     if os.environ.has_key('HTTP_COOKIE'):
          C=Cookie.Cookie(os.environ['HTTP_COOKIE'])
     else:
          C=Cookie.Cookie()
     for eachkey in initialvalues.keys():
          if not C.has_key(eachkey):
               C[eachkey]=initialvalues[eachkey]
          elif C.has_key('studid'):
               C['studid']="S"+str(randint(10,100))
               pass
     return C
if __name__=='__main__':
     cookie=getCookie({'counter':0,'studid':"S01"})
     cookie['counter']=int(cookie['counter'].value)+1
     print cookie
     print "Content-type: text/html\n\n"
     print dynhtml %(cookie['counter'].value, cookie['studid'].value)
```

When you access **hitcount.py** by using a browser, a screen as shown in Figure 14.8 appears.



Figure 14.8 The browser output on first access to hitcount.py.

Close the browser and open it again. When you execute the **hitcount.py** script again, the hit count changes and a different student id is generated, as shown in Figure 14.9.

Let's understand the working of the preceding code. The `getCookie()` function checks the existence of cookies by using the environment variable `HTTP_COOKIE`. When a cookie exists on the client side, which is set by the server on the first visit to a CGI script, requests to the server can be sent using the `HTTP_COOKIE` environment variable. The initial key-value pairs that should be used for the first visit are passed to



Figure 14.9 The browser output on second access to hitcount.py.

getCookie() by using the initialvalues dictionary as a parameter. If some of the cookie values haven't been set, they are added to the cookie by using the initialvalues dictionary. The getCookie() function finally returns a cookie object. The hit count is displayed on the client browser for the first visit by adding 1 to the value of counter. Actually, this is the value that is displayed on the client side that is stored in the cookie. Every time the client accesses this script, the value of counter and, in turn, the value stored in the cookie are increased by one. This is the value that is displayed as the hit count.

After understanding the concept of uploading files and cookies, let's write the code for uploading a weekend assignment for Techsity University.

Write the Code for the CGI Script

Let's write the code for the CGI script.

```
#!/usr/local/bin/python2.2
from cgi import FieldStorage
from os import environ
from cStringIO import StringIO
from urllib import quote, unquote
#from string import strip,split, join
class myCGI:
    header = 'Content-Type: text/html\n\n'
    url = '/cgi-bin/assignmentcgi1.py'
    formhtml = '''<html>
<head>
<title>Tecksity University Assignment Form</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<div align="center">
  <form name="form1" method="post" action="%s"
enctype="multipart/form-data">
    <h1><U>Weekend Assignment Form</U></h1>
    <h2>Student ID:<i> %s </i></h2>
    <table width="400" border="1" bgcolor="#CCCCCC">
      <tr>
        <td width="100"><b>Student ID</b></td>
        <td width="189">
          <input type="text" name="cookie" value="%s">
        </td>
      </tr>
      <tr>
        <td width="100"><b>Student Name</b></td>
        <td width="189">
          <input type="text" name="Stud_Name" value="%s">
        </td>
      </tr>
      <tr>
        <td width="100"><b>Course ID </b></td>
```

```
          <td width="189">
            <input type="text" name="Course_ID" value="%s">
          </td>
        </tr>
        <tr>
          <td width="100"><b>Assignment No.</b></td>
          <td width="189">
            <input type="text" name="Assign_No" value="%s">
          </td>
        </tr>
        <tr>
        <tr>
          <td width="100"><b>Assignment</b></td>
          <td width="189">
            <input type="file" name="File_Upl" value="%s">
          </td>
        </tr>
      </table>  <p>
        <input type="submit" name="Submit" value="Submit">
      </p>
    </form>
    <h1> </h1>
    <h3> </h3>
  </div>
  </body>
  </html>'''

    def FBCookies(self):                # reads cookies from client
        if environ.has_key('HTTP_COOKIE'):
            for eachCook in environ['HTTP_COOKIE'].split(';'):
            eachCook=eachCook.strip()
          if len(eachCook) > 5 and eachCook[:2] == 'FB':
                    tag = eachCook[2:6]
                    try:
                        self.cookies[tag] = eval(unquote(eachCook[7:]))
                    except (NameError, SyntaxError):
                        self.cookies[tag] = unquote(eachCook[7:])
        else:
            self.cookies['info'] = self.cookies['stid'] = ''
        if self.cookies['stid'] != '':
            self.studname,self.courseid,self.assignno,self.f_name \
                = self.cookies['info'].split('$')
        else:
            self.studname = self.f_name = self.courseid=self.assignno=''

    def showForm(self):                        # show fill-out form
        self.FBCookies()
        if not self.cookies.has_key('stid') or self.cookies['stid'] == '':
            cookStatus = studidCook = ''
```

```
        else:
            studidCook = cookStatus = self.cookies['stid']
        print myCGI.header + myCGI.formhtml % \
      (myCGI.url,cookStatus, studidCook, self.studname,\
     self.courseid,self.assignno,self.f_name)
    errhtml = '''<HTML><HEAD><TITLE>
Assignment Submission</TITLE></HEAD>
<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>'''
    def displayError(self):
        print myCGI.header + myCGI.errhtml % (self.error)
    shtml = '''<HTML><HEAD><TITLE>
</TITLE></HEAD>
<BODY>
<h1 align="left"><u>Uploaded Data for Student %s </u></h1>
<H3>Student Name : <B>%s</B></H3>
<H3>Course ID : <B>%s</B></H3>
<H3>Assignment No. : <B>%s</B></H3>
<h2><font face="Georgia, Times New Roman, Times, serif"><u>
<font face="Times New Roman, Times, serif">Uploaded
  file details:</font></u></font></h2>
<h3>File Name: %s</h3>
<H3>Contents:</H3>
<PRE>%s
</PRE>
Click <A HREF="%s"><B>here</B></A> to go back
to the form.
</BODY></HTML>'''
    def setFBCookies(self):
        for eachCook in self.cookies.keys():
            print 'Set-Cookie: FB%s=%s; path=/' % \
                  (eachCook, quote(self.cookies[eachCook]))
    def doResults(self):
        totbytes = 1024
        filedata = ''
        while len(filedata) < totbytes:        # read each line
                                               #from the file
            data = self.f_data.readline()
            if data == '': break
            filedata = filedata + data
        else:                                # truncate if too long
            filedata = filedata + \
                '... <B><I>(file too long to truncated)</I></B>'
        self.f_data.close()
        if filedata == '':
            filedata = '<B><I>(file upload error or file not
supplied)</I></B>'
```

```
        filename = self.f_name
        if not self.cookies.has_key('stid') or self.cookies['stid'] == '':
            cookStatus = '<I>(cookie has not been set yet)</I>'
            studidCook = ''
        else:
             studidCook = cookStatus = self.cookies['stid']
      self.cookies['info'] = '$'.join\
                ([self.studname,self.courseid,self.assignno,filename])
        self.setFBCookies()
        print myCGI.header + myCGI.shtml %\
        (cookStatus, self.studname,self.courseid,\
         self.assignno,filename, filedata, myCGI.url)
    def start(self):                # determine which page to return
        self.cookies = {}
        self.error = ''
        form = FieldStorage()
        if form.keys() == []:
            self.showForm()
            return
        if form.has_key('Stud_Name'):
         val=form['Stud_Name'].value
            self.studname = val.strip().capitalize()
            if self.studname == '':
                self.error = 'Your name is required. (blank)'
        else:
            self.error = 'Your name is required. (missing)'
     if form.has_key('Course_ID'):
         val=form['Course_ID'].value
            self.courseid = val.strip().capitalize()
            if self.courseid == '':
                self.error = 'Course ID is required. (blank)'
        else:
            self.error = 'Course ID is required. (missing)'
     if form.has_key('Assign_No'):
         val=form['Assign_No'].value
            self.assignno = val.strip().capitalize()
            if self.assignno == '':
                self.error = 'Assignment Number is required. (blank)'
        else:
            self.error = 'Assignment Number is required. (missing)'
        if form.has_key('cookie'):
            self.cookies['stid'] = unquote(form['cookie'].value.strip())
        else:
            self.cookies['stid'] = ''
        if form.has_key('File_Upl'):
            File_Upl = form["File_Upl"]
            self.f_name = File_Upl.filename or ''
            if File_Upl.file:
                self.f_data = File_Upl.file
            else:
                self.f_data = StringIO('(no data)')
```

```
        else:
            self.f_data = StringIO('(no file)')
            self.f_name = ''
        if not self.error:
            self.doResults()
        else:
            self.displayError()
if __name__ == '__main__':
    page = myCGI()
    page.start()
```

Execute the CGI Script

To execute the CGI script, perform the following steps:

1. Save the preceding file in `/var/www/cgi-bin` directory as **assignmentcgi.py**.
2. Type the following command:

   ```
   $ chmod +x /var/www/cgi-bin/assignmentcgi.py
   ```

3. In the address bar of Netscape Navigator, enter the following URL:

   ```
   http://localhost/cgi-bin/formresults.py
   ```

4. Assuming that a text file **Assignment.txt** exists in the `/root` directory, enter details in the Web page, as shown in Figure 14.10.

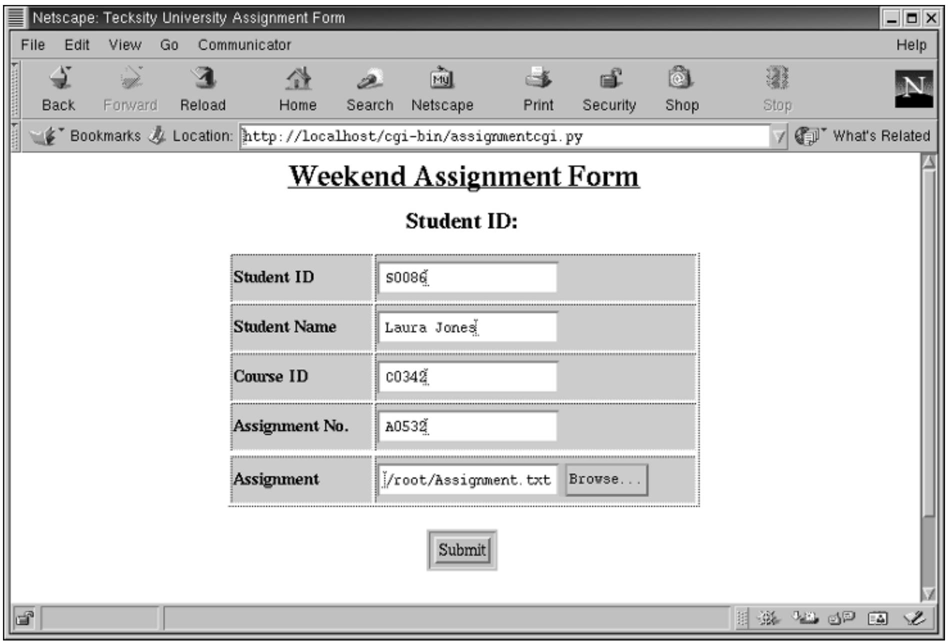


Figure 14.10 Web page with assignment details.

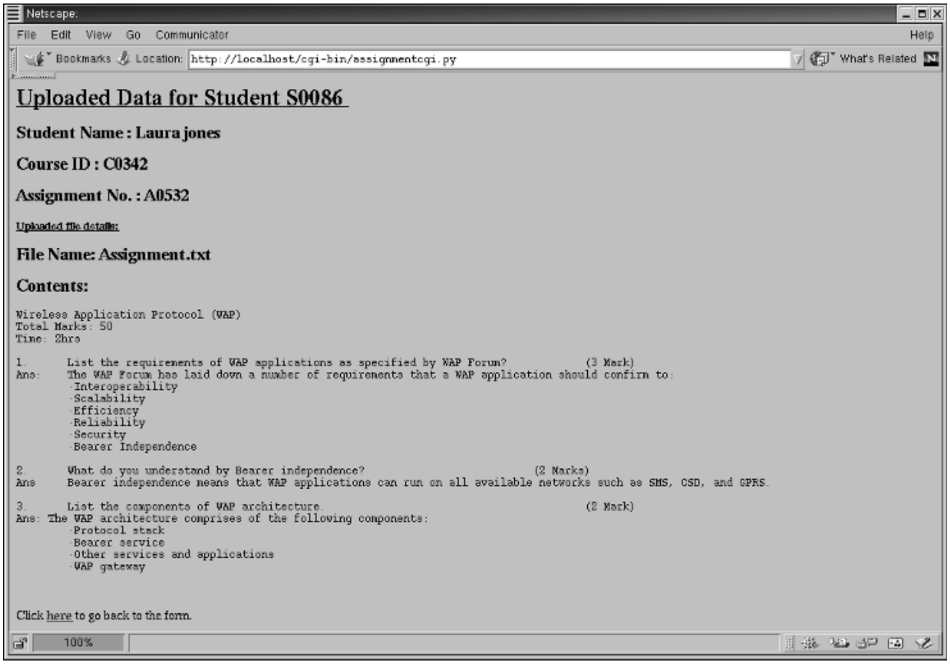


Figure 14.11 Web page showing assignment details and contents of Assignment.txt.

5. Click the Submit button. A page showing the details entered in step 4 appears along with the contents of **Assignment.txt,** as shown in Figure 14.11.
6. Scroll to the bottom of the page, and click the hyperlink denoted by `back`.
7. The Weekend Assignment form appears again, showing the details that were added, as shown in Figure 14.12.

Summary

In this chapter, you learned the following:

- You can create Web servers by using the following modules:

  The `SocketServer` module. Used for creating general IP servers.

  The `BaseHTTPServer` module. Provides the infrastructure required for creating HTTP servers.

  The `SimpleHTTPServer` module. Used for creating simple Web servers.

  The `CGIHTTPServer` module. Used for creating Web servers that support CGI.

- You can use the following modules to process URLs:

  The `urlparse` module. Provides the functionality of manipulating URL strings.

  The `urllib` module. Provides the functionality of retrieving data from the Web by using the given URLs.

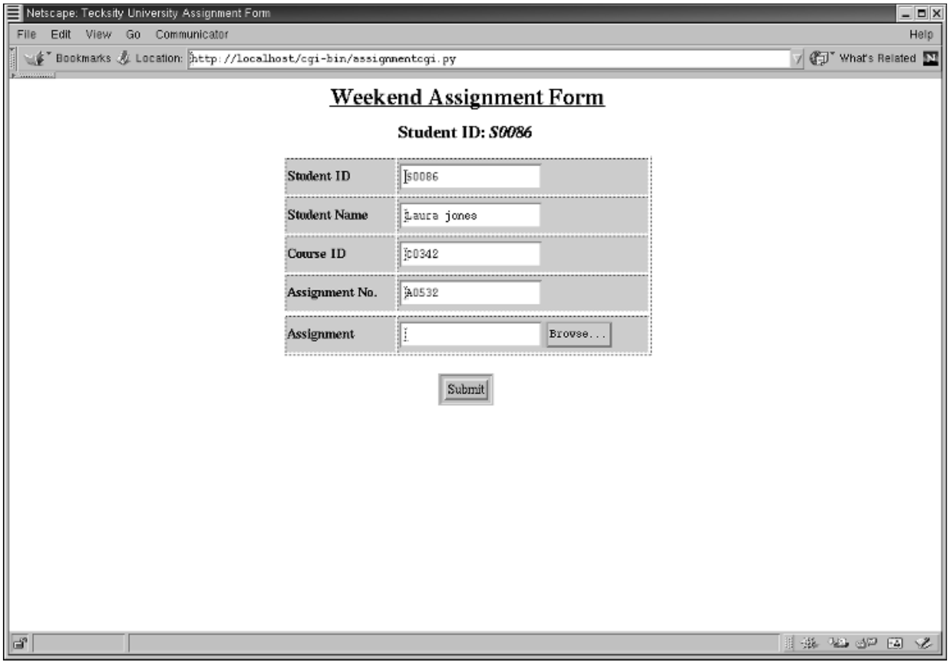


Figure 14.12 Web page showing assignment details after using cookies to extract them.

- A file can be uploaded using the file input type as shown here:

  ```
  <input type="file" name="File_Upload" value="/root/abc.txt">
  ```

- To upload a file to the Web server through a form, form encoding should be specified as `multipart/formdata`.
- The `file` attribute of the uploaded file is used to read data from the uploaded file, and the `filename` attribute of the uploaded file is used to return the name of the file.
- Cookies are pieces of information stored by a Web server on a client computer, and they are managed by a browser.
- The `Cookie` module has many classes that handle the creation of cookie objects. The base class for the creation of all cookie objects is `Cookie`.
- A cookie object contains key-value pairs and behaves like a dictionary. Therefore, a value can be assigned and extracted like a dictionary.

CHAPTER

15

GUI Programming with Tkinter

OBJECTIVES:

In this chapter, you will learn to do the following:

- **Identify the significance of the Tkinter module**
- **Identify the steps to create a GUI application**
- **Identify the widgets provided by the Tkinter module**
- **Use various widgets in your application**

Getting Started

Until now, this book has discussed how to create applications that work on the command-line interface. You execute a Python script and view its output at the Python prompt. If the application requires user input, you enter the input at the prompt. At times, text-based applications can be very monotonous and difficult to work with. This chapter can be helpful for those who want to learn to develop user-friendly graphic interfaces. Imagine how exciting it will be to enable a user to enter the required details in a window with different controls for each detail where the user can activate or choose options by simply pointing or clicking with a mouse instead of asking for

details on the Python prompt. Such applications that interact with a user by means of an interface represented using icons, menus, and dialog boxes on the screen are called *graphical user interface (GUI)* applications.

In this chapter, you will learn about Tkinter, the official GUI framework for Python, to create GUI applications. As a part of this, you will learn about various controls that can be included in a GUI interface. You will further enhance the skills gained in the chapter by designing a GUI application. Before moving on to the concepts related to Tkinter, let's take a brief look at GUI applications.

A graphical user interface (GUI) application, like a painting, has a user interface. In the case of a painting, the canvas holds various components, such as lines, circles, and boxes. Similarly, a GUI application consists of a number of controls, such as text boxes, labels, and buttons, which are contained inside a window. You would have come across a number of GUI applications in day-to-day life. These applications could range from an online registration form on a Web site to a calculator used in home PCs.

Python enables you to create visually appealing GUI applications by using Tkinter. The following section discusses Tkinter.

Introduction to Tkinter

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit. Tkinter provides various controls, such as buttons, labels, and text boxes, used in a GUI application. These controls are commonly called *widgets*.

As mentioned earlier, creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps:

1. Import the `Tkinter` module.
2. Create the GUI application window.
3. Add widgets to the GUI application.
4. Enter the main event loop.

Let's now elaborate on how to perform these steps.

Import the `Tkinter` Module

The `Tkinter` module contains all the classes and widgets required to create a GUI application. To use this module in your application, you need to import it. The following code statement will help you import the `Tkinter` module.

```
import Tkinter           #Statement 1
```

You can also import all the methods, classes, and attributes from the `Tkinter` module by using the following module.

```
from Tkinter import *     #Statement 2
```

Create the Application Window

Any GUI application should first contain a top-level window, or a *root window,* that can further contain the various objects required in the application. The objects contained in the root window could be widgets, such as buttons and labels, or other windows. To create a root window for your application, use the following statement.

```
top = Tkinter.Tk()
```

If you use the `from-import` statement to import all the elements from the `Tkinter` module, then you need to replace the preceding statement with this one:

```
top = Tk()
```

Add Widgets to the Application

Using Tkinter, you can add a number of widgets to your Python application. These widgets can be stand-alone or containers. *Stand-alone* widgets are the ones that do not contain any other widgets, such as a button, a checkbox, and a label. *Container* widgets are the ones that contain other widgets, such as a frame and a window. A container widget is called a *parent* widget, and the contained widgets are called *child* widgets. Various widgets provided by Tkinter are listed in Table 15.1.

Table 15.1 Widgets Provided by Tkinter

WIDGETS	DESCRIPTION
`Button`	The `Button` widget is used to display buttons in your application.
`Canvas`	The `Canvas` widget is used to draw shapes, such as lines, ovals, polygons, and rectangles, in your application.
`Checkbutton`	The `Checkbutton` widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
`Entry`	The `Entry` widget is used to display a single-line text field for accepting values from a user.
`Frame`	The `Frame` widget is used as a container widget to organize other widgets.
`Label`	The `Label` widget is used to provide a single-line caption for other widgets. It can also contain images.
`Listbox`	The `Listbox` widget is used to provide a list of options to a user.

continues

Table 15.1 Widgets Provided by Tkinter (Continued)

WIDGETS	DESCRIPTION
`Menubutton`	The `Menubutton` widget is used to display menus in your application.
`Menu`	The `Menu` widget is used to provide various commands to a user. These commands are contained inside Menubutton.
`Message`	The `Message` widget is used to display multiline text fields for accepting values from a user.
`Radiobutton`	The `Radiobutton` widget is used to display a number of options as radio buttons. The user can select only one option at a time.
`Scale`	The `Scale` widget is used to provide a slider widget.
`Scrollbar`	The `Scrollbar` widget is used to add scrolling capability to various widgets, such as list boxes.
`Text`	The `Text` widget is used to display text in multiple lines.
`Toplevel`	The `Toplevel` widget is used to provide a separate window container.

You will learn to add widgets to your application later in this chapter.

Enter the Main Event Loop

After you design an application by adding appropriate widgets, you need to execute the application. When an application is executed, it enters an infinite loop. This loop includes waiting for an event, such as a mouse-click, processing the event, and then waiting for the next event. The statement that helps your application enter the infinite loop is this:

```
Tkinter.mainloop()
```

If you use the `from-import` statement to import all the elements from the `Tkinter` module, then you need to replace the preceding statement with this one:

```
top.mainloop()
```

In the preceding statement, `top` refers to the top-level window.

Let's put the pieces together and consolidate the code to display a window by using the `Tkinter` module.

```
import Tkinter
top = Tkinter.Tk()
#Code to add widgets
Tkinter.mainloop()
```

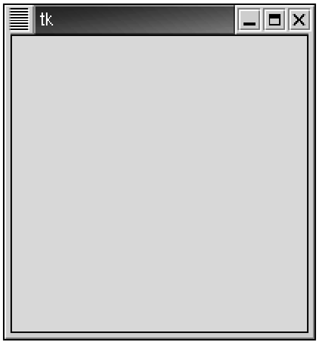


Figure 15.1 A sample window.

You can also rewrite the preceding code as follows:

```
from Tkinter import *
top = Tk()
#Code to add widgets
top.mainloop()
```

The output of the preceding code is shown in Figure 15.1.

Now that you understand the basic steps involved in creating a GUI application by using Tkinter, let's start with creating a GUI application.

Creating a GUI Application

Problem Statement

Techsity University offers a number of courses. Each course has certain prerequisites that a student needs to meet to apply for that course. In addition, a student needs to be 21 years of age to be eligible for applying. The online site of Techsity University has a way to inform students about the prerequisites of a course. All that a student needs to do is fill out an online form. This form requires the details of the course selected and displays the prerequisites for the course. This form also indicates whether a course is offered part time.

You, as a developer, have been assigned the task of designing a form and writing the code that performs the required job. The code, when executed, will display a form that prompts a student to provide certain input. The result is then displayed in a message box when a button is clicked. Moreover, a student is addressed appropriately in the message box—for example, Mr. Tom Smith. Let's identify the tasks needed to create this application. As an add-on, you need to create buttons that clear all the widgets present in the window and close the window.

Task List

- ✓ **Identify the components of the user interface of the form.**
- ✓ **Identify the Tkinter elements to design the user interface.**
- ✓ **Write the code for the user interface.**
- ✓ **Execute the code.**

Identify the Components of the User Interface

A form has to be designed as a user interface to gather the required information from a student:

- Personal details

  The user's name that can be split into two parts, first name and last name

  The age of the student

  The gender of the student
- Course details

  The course selected

  Whether the course is offered part time

Identify the Tkinter Widgets to Design the User Interface

Table 15.2 describes the Tkinter widgets to be used for the design of the form.

Table 15.2 Widgets to Be Used in the Window

WIDGET	PURPOSE
`Label`	To provide captions for various other widgets.
`Entry`	To display a single-line entry field for accepting values, such as the first name and the last name.
`Listbox`	To display a list of available courses.
`Radiobutton`	To accept the gender of a student.
`Checkbutton`	To provide the option of checking the availability of a part-time course.
`Button`	To display the prerequisite for the opted course, clear the widgets, and close the window.
`Frame`	To organize various radio buttons and buttons.

Let's now look at the details of these components.

The `Label` Widget

The `Label` widget is used to display text or provide captions for other widgets. For example, you can use a label to provide captions for various other widgets present in a window. In addition, you can display bitmaps and images in a label. Use the following syntax to display a text label in a window.

```
L1 = Label(top, text="Hello World")
```

In the preceding code,

- `top` refers to the window on which the label is to be displayed.
- `text` option is used to specify the text to be displayed in the label.

You can also specify the width and height of a label by using the `width` and `height` options, respectively. The statement to do so is as follows:

```
L1 = Label(top, text="Hello World", width = 20, height =5)
```

The complete code that creates a label in a window (as shown in Figure 15.2) is as follows:

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="Hello World", width = 20, height =5)
L1.pack()
top.mainloop()
```

If the label displays text, then the unit of measurement is text units. If the label displays an image, then the unit of measurement is pixels.

Table 15.3 lists some other options that you can use with the Label widget.



Figure 15.2 A sample window displaying a label.

Table 15.3 Various Options of the Label Widget

OPTION	DESCRIPTION
`bitmap`	`bitmap` specifies the bitmap to be displayed.
`borderwidth`	`borderwidth` specifies the width of the label border.
`bg`	`bg` specifies the background color of the label.
`fg`	`fg` specifies the color of the text present in the label.
`font`	`font` specifies the font of the text to be displayed.
`justify`	`justify` specifies the alignment of multiple lines of text. Various possible values are `LEFT`, `RIGHT`, or `CENTER`.

The `Entry` Widget

The `Entry` widget is used to accept single-line text strings from a user. Let's now look at the syntax to display an `Entry` widget in your application.

```
E1 = Entry(top)
```

Like the `Label` widget, you can use various options with the `Entry` widget. Some of these options are listed in Table 15.4.

The following code statement implements some of the options of the `Entry` widget.

```
E1 = Entry(top, bd =5, fg = "red", relief = RAISED)
```

The preceding statement displays a text field in the `top` window. The border width of this window is five pixels, and the color of the text is red.

Table 15.4 Various Options of the Entry Widget

OPTION	DESCRIPTION
`bd`	`bd` specifies the width of the Entry widget border.
`bg`	`bg` specifies the background color of the Entry widget.
`fg`	`fg` specifies the color of the text in the Entry widget.
`font`	`font` specifies the font of the text in the text field.
`relief`	`relief` specifies the type of the border. Some of the values are `SUNKEN`, `RAISED`, `GROOVE`, and `RIDGE`.



Figure 15.3 A sample window displaying an `Entry` widget.

Following is the complete code to display an `Entry` widget:

```
from Tkinter import *
top = Tk()
E1 = Entry(top, bd =5, fg = "red", relief = RAISED)
E1.pack()
top.mainloop()
```

This field appears raised (as shown in Figure 15.3).

In addition to these options, an `Entry` widget also provides a number of methods. Table 15.5 lists some of these methods.

Table 15.5 Various Methods to Manipulate the Entry Widget

METHOD	FUNCTION	EXAMPLE
`insert(index, text)`	This method inserts text at the given index. Some of the values used to specify index are `INSERT` and `END`.	`E1.insert (INSERT, "Hello")` This statement inserts `Hello` at the current cursor position.
`delete(index)`	This method deletes the character at the specified index.	`E1.delete(1)` This statement deletes the character at the index position `1`.
`delete(from, to)`	This method deletes the characters within the specified range.	`E1.delete(0, END)` This statement deletes all the characters present in a string.
`get()`	This method retrieves the contents present in the text field.	`E1.get()` This statement returns the contents of the `E1` Entry widget.

Before proceeding further, let's look at a code sample to display an entry field for a username along with a suitable caption.

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
E1 = Entry(top, bd =5)
top.mainloop()
```

When you execute the preceding code, the corresponding window that appears does not display any control in the window because you did not arrange the widgets in the parent window. Tkinter provides you with various classes that help you organize the placement of widgets in a window. These classes are also called *geometry managers*.

Geometry Managers

Widgets in a window should be in a proper layout so that they do not appear scattered. For this purpose Tkinter provides a powerful concept called *geometry management*. Geometry management is the technique used to organize widgets in their container widget. Tkinter provides a powerful and flexible model to manage the placement of widgets in a container.

To organize various widgets inside a window or another widget, Tkinter provides three classes or geometry managers: `pack`, `grid`, and `place`. The following list describes these classes.

- The `pack` geometry manager organizes widgets in rows or columns inside the parent window or the widget. To manage widgets easily, the `pack` geometry manager provides various options, such as `fill`, `expand`, and `side`.

  `fill.` The `fill` option is used to specify whether a widget should occupy all the space given to it by the parent window or the widget. Some of the possible values that can be used with this option are `NONE`, `X`, `Y`, or `BOTH`. By default, the `fill` option is set to `NONE`.

  `expand.` The `expand` option is used to specify whether a widget should expand to fill any extra space available. The default value is `zero`, which means that the widget is not expanded.

  `side.` The `side` option is used to specify the side against which the widget is to be packed. Some of the possible values that can be used with this option are `TOP`, `LEFT`, `RIGHT`, and `BOTTOM`. By default, the widgets are packed against the `TOP` edge of the parent window.

Let's now rewrite the code to display a `Label` and an `Entry` widget that we discussed in the previous section by using the `pack` class.

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.pack(side=LEFT)
E1 = Entry(top, bd =5)
E1.pack(side=RIGHT)
top.mainloop()
```



Figure 15.4 Organizing widgets by using the `pack` geometry manager.

When you execute the preceding code, a window containing both the widgets appears, as shown in Figure 15.4.

- The `grid` geometry manager is the most flexible and easy-to-use geometry manager. It logically divides the parent window or the widget into rows and columns in a two-dimensional table. You can then place a widget in an appropriate row and column format by using the `row` and `column` options, respectively. To understand the use of `row` and `column` options, consider the following code.

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.grid(row=0, column=0)
E1 = Entry(top, bd =5)
E1.grid(row=0, column=1)
top.mainloop()
```

When you execute the preceding code, a window containing both the widgets appears, as shown in Figure 15.5.

- The `place` geometry manager allows you to place a widget at the specified position in the window. You can specify the position either in absolute terms or relative to the parent window or the widget. To specify an absolute position, use the `x` and `y` options. To specify a position relative to the parent window or the widget, use the `relx` and `rely` options. In addition, you can specify the size of the widget by using the `width` and `height` options provided by this geometry manager.

Let's now look at the code to implement the place geometry manager.

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.place(relx=0.0, rely=0.0)
E1 = Entry(top, bd =5)
E1.place(relx=0.4, rely = 0.0)
top.mainloop()
```

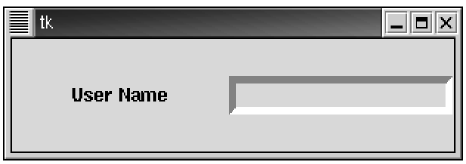


Figure 15.5 Organizing widgets by using the `grid` geometry manager.

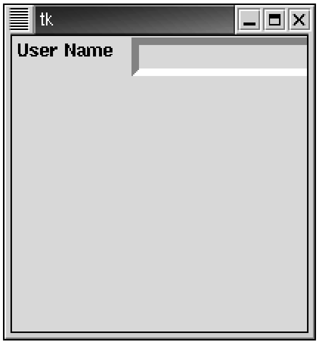


Figure 15.6 Organizing widgets by using the `place` geometry manager.

When you execute the preceding code, a window containing both the widgets appears, as shown in Figure 15.6.

NOTE While using the `relx` and `rely` options, 0.0 refers to the upper left edge and 1.0 refers to the lower right edge.

The `Button` Widget

The `Button` widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button, which is called automatically when you click the button. Consider the following statement that is used to display a button.

```
self.w=Button(top, text ="Say Hello", command=self.Call_Hello)
```

In the preceding code,

- `top` represents the parent window.
- The `text` option is used to specify the text to be displayed on the button.
- The `command` option is used to specify the function or procedure that is called when a user clicks the button. In this case, the `Call_Hello()` method is called.

Table 15.6 lists some of the options that can be used with the `Button` widget.

Table 15.6 Various Options of the Button Widget

OPTION	DESCRIPTION
bg	`bg` specifies the background color of the button.
fg	`fg` specifies the color of the text in the button.
font	`font` specifies the font of the text.

OPTION	DESCRIPTION
relief	relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
image	image specifies the image to be displayed in the button.
width, height	width and height specify the size of the button.

Let's now look at a code snippet that displays a button and then displays a message to say hello to the user.

```
import Tkinter
import tkMessageBox
top = Tkinter.Tk()
def hello():
    tkMessageBox.showinfo("Say Hello", "Hello World")
B1 = Tkinter.Button(top, text = "Say Hello", command = hello)
B1.pack()
top.mainloop()
```

When you execute the preceding code, a window containing a button appears, as shown in Figure 15.7. Next, you click the Say Hello button, and a message displaying Hello World appears.

You would have noticed that in the preceding code, we used a module called tkMessageBox. The following section discusses the details of this module.

The tkMessageBox Module

The tkMessageBox module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message. Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore. The syntax to display a message box is this:

```
tkMessageBox.FunctionName(title, message [, options])
```

In the preceding code,

- FunctionName is the name of the appropriate message box function.
- ttitle is the text to be displayed in the title bar of a message box.



Figure 15.7 A sample window containing a button.

- `message` is the text to be displayed as a message.
- `options` are alternative choices that you may use to tailor a standard message box. Some of the options that you can use are default and parent. The `default` option is used to specify the default button, such as `ABORT`, `RETRY`, or `IGNORE` in the message box. The `parent` option is used to specify the window on top of which the message box is to be displayed.

NOTE **Before using the `tkMessageBox` module, you need to import it by using the following statement:**

```
import tkMessageBox
```

The `Listbox` Widget

The `Listbox` widget is used to display a list of items from which a user can select a number of items. To create a list box in your application, use the following syntax.

```
Lb1 = Listbox(top)
```

The preceding code creates a blank list box, as shown in Figure 15.8. Therefore, you need to add items to it. To do so, you use the `insert` method. The syntax of this method is described here.

```
Lb1.insert(index, item)
```

In the preceding syntax,

- `index` refers to the index position at which an item is to be inserted. Some of the possible values of an index are `INSERT` and `END`. The `INSERT` value places the item at the current cursor position, and the `END` value places the item at the end.
- `item` refers to the value to be inserted. Item can be of the text type only.

For example,

```
Lb1.insert(END, "Rose")
```

The preceding statement inserts the item `Rose` at the end of the `Lb1` listbox.

Let's now write a complete code to insert a listbox in a window.

```
from Tkinter import *
import tkMessageBox
top = Tk()
Lb1 = Listbox(top)
Lb1.insert(1,"Python")
Lb1.insert(2,"Perl")
Lb1.insert(3,"C")
Lb1.insert(4,"PHP")
Lb1.insert(5,"JSP")
Lb1.insert(6,"Ruby")
Lb1.pack()
top.mainloop()
```

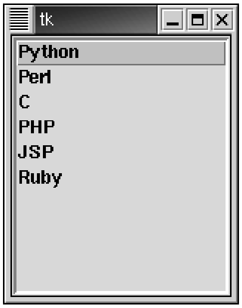


Figure 15.8 A Window containing the `Listbox` widget.

The preceding code creates a `Listbox` widget containing the names of different languages at the specified indices, as shown in Figure 15.8.

The `Listbox` widget provides a number of other methods that ease your working with this widget. Some of these methods are listed in Table 15.7.

Table 15.7 Methods Provided by the `Listbox` Widget

METHOD	FUNCTION	EXAMPLE
`curselection()`	This method retrieves the index position of the selected index.	`Lb1.curselection()` This statement returns the index position of the currently selected item.
`delete(index)`	This method deletes the item at the specified index.	`Lb1.delete(1)` This statement deletes the item at the index position `1`.
`delete(first, last)`	This method deletes the items within the specified range. For example, you can use `0`, `END` to delete all the items in the list.	`Lb1.delete(0, END)` This statement deletes all the items present in the list box.
`get(index)`	This method retrieves the item present at the specified index.	`E1.get(1)` This statement returns the item present at the index position `1` of the list box.



Figure 15.9 A window containing the `Checkbutton` widget.

The `Checkbutton` *Widget*

The `Checkbutton` widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option. You can also display images in place of text. The syntax to display a check button in an application is this:

```
CheckVar = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar)
```

In the preceding syntax,

- `top` refers to the parent window.
- The `text` option specifies the text to be displayed.
- The `variable` option attaches a Tkinter variable (`CheckVar`) to the check button. When you click the button, the value contained in the variable is toggled between the on value and the off value, which specifies whether the button is checked or unchecked. You can set these values by using the `onvalue` and `offvalue` options.

Let's write the code to display a `Checkbutton` widget in a window.

```
from Tkinter import *
import tkMessageBox
top = Tkinter.Tk()
CheckVar = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar, \
onvalue = 1, offvalue = 0)
C1.pack()
top.mainloop()
```

The preceding code creates a check button, Music, as shown in Figure 15.9. Table 15.8 lists some of the methods that you can use with a check button.

Table 15.8 Methods Provided by the Checkbutton Widget

METHOD	FUNCTION	EXAMPLE
`deselect()`	To deselect the button	`C1.deselect()`
`select()`	To select the button	`C1.deselect()`
`toggle()`	To reverse the toggle state of the button	`C1.toggle()`

The `Radiobutton` Widget

Like the `Checkbutton` widget, the `Radiobutton` widget is also used to display a number of options to a user as toggle buttons. A user can select only one option at a time, though. The syntax to display a radio button is this:

```
from Tkinter import *
import tkMessageBox
top = Tkinter.Tk()
RadioVar = IntVar()
R1 = Radiobutton(top, text = "Male", variable = RadioVar, value = 1)
R1.pack()
R2 = Radiobutton(top, text = "Female", variable =RadioVar,value = 2)
R2.pack()
top.mainloop()
```

The preceding code creates two radio buttons, `Male` and `Female`, as shown in Figure 15.10. You need to add these buttons to one group so that a user can select only one of them at a time. To do so, ensure that the `variable` option points to the same variable name (`RadioVar`).

Like the `Checkbutton` widget, a `Radiobutton` widget also supports `select()` and `deselect()` methods. These methods are used to select and deselect the button, respectively.

The `Frame` Widget

The `Frame` widget is a container widget used to organize other widgets. Frame refers to a rectangular area on a parent window. To understand the use of the `Frame` widget, consider a situation in which you need to add a number of radio buttons to your application. Organizing a large number of radio buttons in the parent window is a tedious task. Therefore, to simplify this process, you can add all the radio buttons to a frame and then add the frame to the parent window. The syntax to create a frame is this:

```
F1 = Frame(top, width = 100, height = 100)
```

The preceding code creates a frame of the size that is specified using the `width` and `height` options. This frame is created in the `top` window.

The following code demonstrates the process of adding widgets to a frame.

```
r1=Radiobutton(F1, text="Male", variable=v, value=1)
r2=Radiobutton(F1, text="Female", variable=v, value=2)
```

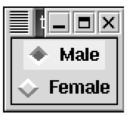


Figure 15.10 A window containing the `Radiobutton` widget.

Write the Code for the User Interface

After identifying the widgets required to design the user interface, let's write the code for the user interface to display the prerequisites of a course.

```
from Tkinter import *
import tkMessageBox
class App:
    def __init__(self, master):
        #First Name
        Label(master, text="First Name").grid(row=0)
        self.e1=Entry(master)
        self.e1.grid(row=0, column=1)
        #Last Name
        Label(master, text="Last Name").grid(row=1)
        self.e2=Entry(master)
        self.e2.grid(row=1, column=1)
        #Age
        Label(master, text="Age").grid(row=2)
        self.e3=Entry(master)
        self.e3.grid(row=2, column=1)
        #Blank
        Label(master, text="", width=5).grid(row=0, column=3)
        #Gender
        Label(master, text="Gender").grid(row=0, column=4)
        self.f1=Frame(master, relief= "sunken", bd=2)
        self.v=IntVar()
        self.r1=Radiobutton(self.f1, text="Male",\
        variable=self.v, value=1).pack(anchor=W)
        self.r2=Radiobutton(self.f1, text="Female",\
        variable=self.v, value=2).pack(anchor=W)
        self.f1.grid(row=1, column=4)
        #Blank
        Label(master, text="").grid(row=3)
        #Course Applied For
        Label(master, text="Course Applied for:",
wraplength=60).grid(row=4)
        self.L1 = Listbox(master, width = 25, height = 4)
        for item in ["Quality Management (Adv.)",\
       "Financial Management (Adv.)",\
       "Project Management (Adv.)",\
       "Project Management (Int.)"]:
            self.L1.insert(END, item)
        self.L1.grid(row=4, column=1)
        #Buttons
        self.f2=Frame(master)

        self.w=Button(self.f2, text ="Prerequisites", height =1,\
     width=10, command=self.Chk_Prereq, default=ACTIVE).pack()
        self.w1=Button(self.f2, text ="Clear", height =1, \
    width=10, command=self.Clear).pack()
```

```
        self.w2=Button(self.f2, text ="Cancel", height=1, \
    width=10, command=self.Close).pack()
        self.f2.grid(row=4, column=4)
         #Blank
        Label(master, text="").grid(row=6)
        #Checkbox
        self.var=IntVar()
        self.c=Checkbutton(master, text="Part-Time Course", variable=
self.var, offvalue=0, onvalue=1)
        self.c.grid(row=7)

    def Chk_Prereq(self):
        self.Eval()
    def Eval(self):
        self.fname = self.e1.get()
        self.lname = self.e2.get()
        self.age =  int(self.e3.get())
        #Check for Age
        if self.age < 21:
            tkMessageBox.showwarning("Invalid Age",\
 "You are not eligible")
            return
        #Check for Gender
        if self.v.get()==1:
            self.str1 = "Dear Mr."
        elif self.v.get()==2:
            self.str1 = "Dear Ms."
        else:
            tkMessageBox.showwarning("Missing Info", \
           "Please select the appropriate gender")
            return
        #Check for Prereq Course
        self.name = self.str1 + " " + self.fname + " " + self.lname
        self.varl1 = self.L1.get(self.L1.curselection())

        if self.varl1 == "Quality Management (Adv.)":
            self.prereq =
          "The prereq for this course is Quality Management (Int)."
            self.flag = 1
        elif self.varl1 == "Financial Management (Adv.)":
            self.prereq = \
    "The prereq for this course is Financial Management (Bas)."
            self.flag = 1
        elif self.varl1 == "Project Management (Adv.)":
            self.prereq = \
           "The prereq for this course is Project Management (Int)."
            self.flag = 0
        else:
            self.prereq = \
           "The prereq for this course is Project Management (Bas)."
            self.flag = 0
```

```
        #Check whether Part Time
        if self.var.get() == 1 and self.flag == 0:
            self.str2 = "\nThis course is not available part time."
        elif self.var.get() == 1 and self.flag == 1:
            self.str2 = "\nThis course is available part time."
        else:
            self.str2 = ""
        self.result = self.prereq + self.str2
        tkMessageBox.showinfo(self.name, self.result)
    def Close(self):
        root.destroy()

    def Clear(self):
        self.e1.delete(0,END)
        self.e2.delete(0,END)
        self.e3.delete(0,END)
        self.c.deselect()
        self.L1.select_clear(self.L1.curselection())

root = Tk()
app = App(root)
root.mainloop()
```

Execute the Code

To be able to implement or view the output of the code to design the user interface and display the prerequisites of a course, you need to execute the following steps:

1. Save the file as **DispPrereq.py**.
2. At the shell prompt, type `python` followed by the name of the file if the file is in the current directory. A window appears, as shown in Figure 15.11.

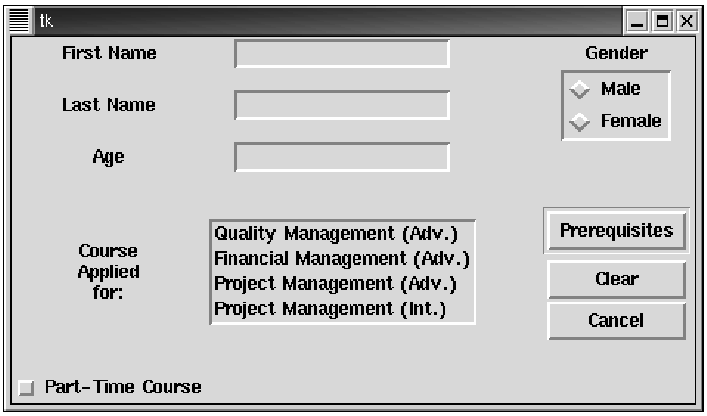


Figure 15.11 Techsity University—the prerequisites form.



Figure 15.12 The `Missing Info` message box.

3. In the window that appears, enter the following details:

   First Name: John

   Last Name: Smith

   Age: 21

   Course Applied For: Quality Management (Adv.)

4. Click the Prerequisites button. A message box appears, as shown in Figure 15.12. Close the message box.
5. Click the Clear button. The contents of all the widgets are deleted.
6. Repeat step 3 with the following modifications:

   Gender: Male

   Age: 20

7. Click the Prerequisites button. A message box appears, as shown in Figure 15.13.
8. Repeat step 3 with the following modifications:

   Gender: Male

   Age: 21

9. Click the Prerequisites button. A message box appears, as shown in Figure 15.14. Close the message box.



Figure 15.13 The `Invalid Age` message box.

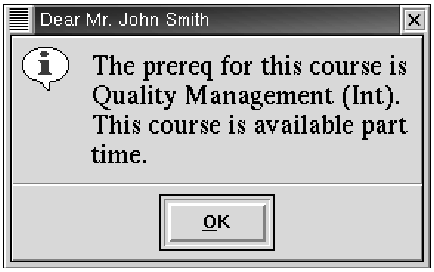


Figure 15.14 The message box displaying the prerequisite.

10. Click the Cancel button. The window is closed.

Summary

In this chapter, you learned the following:

- The Tkinter module is a collection of classes that help you create GUI applications in Python.
- The steps involved in creating a GUI application by using Tkinter are as follows:
  1. Import the `Tkinter` module.
  2. Create the application window.
  3. Add widgets to the application.
  4. Enter the main event loop.
- The `Label` widget is used to display text.
- The `Entry` widget is used to accept single-line text strings from a user.
- The `Button` widget is used to display various types of buttons.
- The `Listbox` widget is used to display a list of items from which a user can select one or more items.
- The `Checkbutton` widget is used to display a number of options to a user as a toggle button. A user can select more than one option by clicking the button corresponding to an option.
- The `Radiobutton` widget is also used to display a number of options to a user as toggle buttons; however, a user can select only one option at a time.
- The `Frame` widget is the container widget that is used to organize other widgets.

APPENDIX

A

Distributing COM Objects

Programming languages such as Visual Basic, C++, Java, and Delphi have comparable strengths and weaknesses when used in a development environment. A major drawback, though, is that an application or a component written in a specific programming language is unable to communicate or use the functionality of a component written in another language. For example, you create the functions to perform calculations in Python. You work on a Visual Basic application for which you need a calculator. Now, how will you use the calculation functions created in Python in your Visual Basic application? COM is the answer to this question. Component Object Model (COM) creates a cross-platform bridge for objects written in different languages to communicate with each other.

Basics of COM

Component Object Model (COM) is a software architecture that allows applications and systems to be built from the components supplied by different software vendors. It is a set of binary and network standards that allows software applications to communicate with each other regardless of the hardware, operating system, and programming language used for development.

A component is a program, or a binary object, that performs a specific operation. In a component-based system, the components interact by calling methods and passing

data. COM ensures that there is a standard method of interaction between the components. All COM objects need to follow these standards when providing functionality. COM is not a programming language. It is a specification that defines how components can communicate with each other.

One of the most important features of COM is the COM specification. The COM specification is a standard that describes the appearance and behavior of COM objects and is a set of services. The COM library provides these services as part of the operating system in a Win32 platform and as a separate package for other operating systems. COM defines the standards in which objects or components are defined within applications and other software components. These software objects are shared so that other objects and applications can use them. Several services and applications in the Windows platform follow and support COM specifications. Although COM standards are applied in a large number of cases in Windows, it cannot be generalized that COM is not supported by other platforms. COM is platform-independent, and COM components can be created in any language and operating system if they adhere to the COM specifications.

Some of the other features of COM are as follows:

COM is object oriented. COM components are true objects in the usual sense—they have an identity, state, and behavior.

COM enables easy customization and upgrades to your applications. COM components link dynamically with each other, and COM defines standards for locating other components and identifying their functionality. Therefore, components can be swapped without having to recompile the entire application.

COM enables distributed applications. Location transparency is one of the features of COM. This enables you to write applications regardless of the location of the COM components they use. The components can be moved without requiring any changes to the application using them.

COM components can be written using many languages. Any language that can handle a binary standard can be used to create COM components. You can use Python, C, C++, Java, Visual Basic, and Visual C++ languages to create components.

COM-based components are self-versioning. COM provides a facility for version control. This implies that new functionality can be added to a component without affecting the clients that already use the component.

COM provides access to functionality of components through interfaces. A COM component is a stand-alone and encapsulated object. So that other objects can access functionality provided by the component, every component has an interface. This interface defines the behavior of the component. A client application has to interact with the interface to call the methods of the component.

COM components can also be created using Python. The support for COM in Python is available in the form of `win32com` Python extensions. These extensions are installed as a package when you run the `win32all` executable file to install the PythonWin distribution. Therefore, you can use the `win32com` package to create COM components that can communicate with other COM-aware applications, such as Microsoft Word, Microsoft Excel, and Visual Basic.

The Binary Standard

In Component Object Model, components are pieces of code that are capable of communicating with each other. Component objects can be implemented in a number of programming languages and can be used for client software programs written in different languages. COM provides a binary standard for communication between objects. This is done in COM through virtual function tables. This concept of virtual function table is borrowed from C++. COM lays down specifications for loading the virtual table in memory and provides a standard way to access functions from the virtual table. All languages that support calling of functions through pointers can be used to create components that interoperate with the components written in any other language but following the same binary standard. For example, an object or component written in Visual Basic can communicate and share data with an object written in Python if the objects follow the COM binary standard.

Each COM component is written to meet the binary standards set by COM. Some of these standards are as follows:

- Components need to keep track of their own creation and destruction.
- Components need to provide their functionality through a standard procedure.
- The location of the component needs to be transparent to the client.

Each component has a unique ID for its identification. These IDs are present in the HKEY_CLASSES_ROOT hive of the system registry in Windows. Figure A.1 shows the registry entry for Microsoft Internet Explorer.

From the previous discussion, you have learned that COM can be used to create reusable components that connect to form applications. The interaction between components in COM is based on the client/server model. Based on this model, COM components can be categorized as follows:

Client components. Components using the functionality provided by other components.

Server. COM components with a predefined functionality that other components can use.

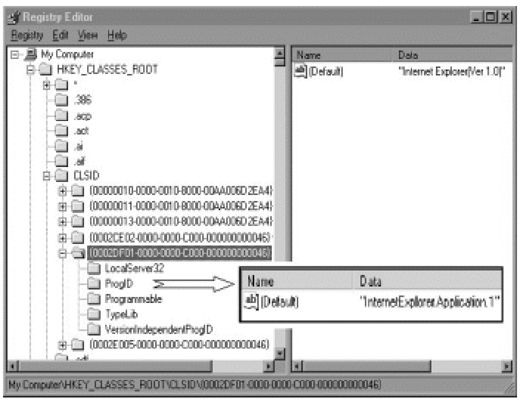


Figure A.1 Registry view.

Consider a situation in which a user wants to insert a bitmap image into a Microsoft Word document. The Object dialog box that is displayed (refer to Figure A.2) when the Object option is selected from the Insert menu option displays the COM components available on the system. These components will fall into the server category.

In this example, the Word document requires access to the functionality provided by the bitmap image. The Word document is acting like a client.

Consider another situation in which a user wants to insert a Word document into an Excel worksheet. Here, the Word document is acting as a server and the Excel worksheet as a client. The Excel worksheet is accessing the functionality of the Word application.

Another feature of COM components is that COM is designed to allow components to communicate with each other regardless of their location. For example, the user interface component would be best located on the same computer as the client. Alternatively, a component that supplies statistical calculations of remote data would most likely be located on a separate computer with the data to be manipulated.

COM server components can be categorized into two types:

In-process server. In-process servers are ideal for situations that involve the transfer of large amounts of data between the server and the client component. In-process servers occupy the same address space as the client application; therefore, they can communicate with the client at a faster speed. In-process servers are implemented as Dynamic Link Libraries (DLLs). DLLs allow specific sets of functions to be stored separately from the executable as files with the DLL extension. These DLLs are loaded into the memory only when the program requires them.

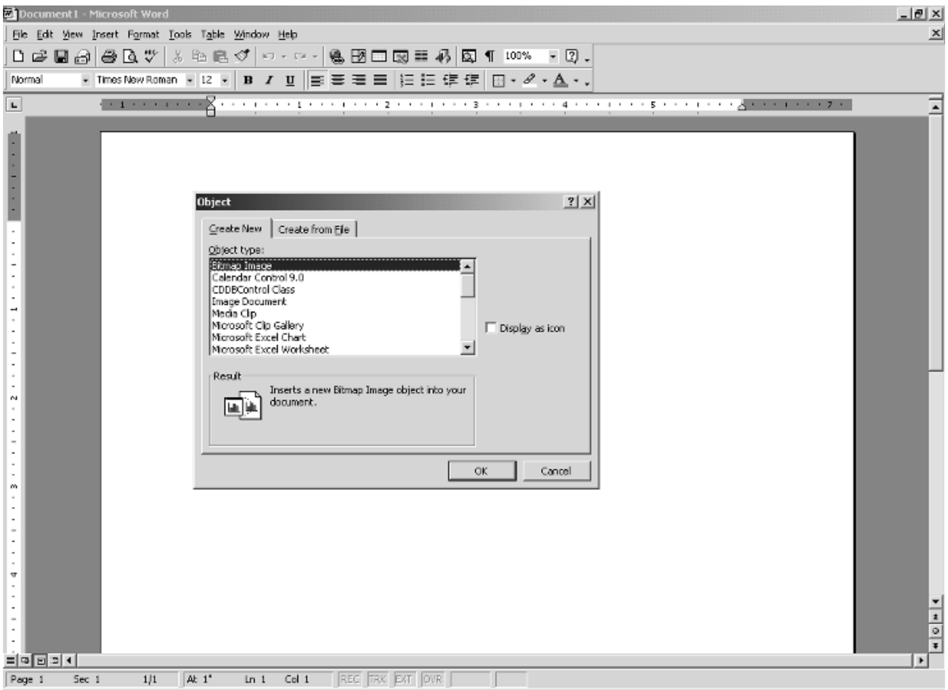


Figure A.2 Inserting a bitmap image into a Word document.

Out-process server. Out-process servers are ideal for the components that need to run either in a separate process space or in a thread separate from the client application. These servers are a little slower because the data has to be moved from one address space to another. Out-process servers are implemented as stand-alone EXEs and run in their own threads. The clients do not block the servers when the client code is executing.

When you create a COM component, how does it interact with client applications? The answer to this lies in COM interfaces. The following section provides a brief description of COM interfaces.

COM Interfaces

COM components are highly encapsulated. The process of internal implementation of the component object is hidden from the user. There is no way to know the data structures that the object uses and how these data structures are manipulated by the functions. If the component object is well encapsulated, how does it communicate with other objects? The answer to this question lies in an interface. The only way to access a COM object is through an interface. *Interfaces* are groups of functions that are used by the applications to interact with each other and with the operating system.

Interfaces are only the declarations of these functions, and the interfaces do not carry any implementations of these functions. These functions have to be implemented in the component. To call the methods of the component object, a client application will have to create a pointer to the interface of the component object. Using the pointer, the client application will be able to call the methods of the component object. Let's understand how this happens.

Each interface provides one or more functions that another object or program can call. Each interface is uniquely identified by its Globally Unique Identifier (GUID). A GUID is a 128-bit or 16-byte number that can be assigned to an interface, a class, or a library. GUIDs ensure that there are no naming conflicts and that COM components do not accidentally connect to the wrong component. A developer creating an interface also needs to create an identifier for that interface. By doing this, any naming conflicts that can arise during run time are eliminated. If a new functionality is added to an interface, a new GUID is assigned to the interface. This makes an interface immutable. Therefore, no versioning of an interface is required.

To create an object, COM locates an object and creates its instance. COM classes are similar to other Python classes. Each COM class needs to implement the following types of GUIDs:

- Interface ID (IID) is a GUID for uniquely identifying an interface.
- Class ID (`_reg_clsid_`) is a GUID for uniquely identifying a class.
- Program ID (`_reg_progid_`) is a user-friendly name for a class ID.

COM provides a set of standard interfaces such as `IUnknown`, `IDispatch`, `IStream`, `Istorage`, and `IOle`. The `IUnknown` interface is the base interface that has to be implemented by all the other interfaces. The `IUnknown` interface contains three methods: `AddRef()`, `Release()`, and `QueryInterface()`. Both `AddRef()` and `Release()` manage the lifetime of a COM component. `AddRef()` increments a

counter that counts the number of clients accessing the component. `Release()` decrements the counter when a client stops using the component object. `QueryInterface()` is used to obtain the other interfaces that the object exposes. It provides a pointer to the functions of the interface. The `IDispatch()` interface is used by the component to interact with other components.

To access a property or method of a component, the object has to be bound to the component property or method. This is called binding. Let's see how Python uses binding to access properties and methods of COM objects.

Binding

The process of associating a function to an object is called *binding*. When the type of object to be created is known at the time of compilation, a particular function of the COM object can be associated with the calling object. This process is called *early binding*.

In certain situations, the type of the object may not be known at the time of compilation. Consider a class hierarchy, which represents the family of personal computers (PC), and the need to create a list of personal computers . Depending on the request of a user, the attributes of a particular computer, such as the speed of the CPU, the hard disk capacity, and the memory space, may have to be displayed. The objects are dynamically created. Therefore, the types of objects are not known at the time of compilation. In these situations, the binding of the function to the object is done at run time. This process is called *dynamic* or *late binding*.

The Python interpreter does not know the properties and methods exposed by an object. Therefore, it uses the concept of late binding to access the methods and properties of the object. The Python interpreter associates with the methods and properties of the object at run time. Late bindings use the `IDispatch` interface to dynamically determine the object model. The `client.dispatch()` function provided by the `win32com` module implements late bindings. The names are converted internally into IDs by using an internal function, `GetIDsOfNames()`. Then, the ID generated is passed internally to the `Invoke()` function.

This type of association, which happens at run time, can be time-consuming because names are resolved at run time. The performance can be improved by early binding. In the case of early binding, names are not resolved at run time, and the object model is exposed at compile time. In this type of implementation, methods and properties of the object cannot be called directly. You will have to use the `GetIDsOfNames()` method, which returns an ID for the method or property of the object that you want to use, and the invoke method, which calls the property or the method. For example, you can invoke a function call as follows:

```
id= GetIDsOfNames("MethodCall")
Invoke(id,DISPATCH_METHOD)
```

You can invoke a function call as follows:

```
id= GetIDsOfNames("Property")
Invoke(id,DISPATCH_METHOD)
```

You can also call the function as follows:

```
MyObject.MethodCall()
```

In addition, you call the property as follows:

```
MyObject.ObjectProperty
```

Python and COM

The support for COM in Python is provided by two packages:

The `pythoncom` extension module. The `pythoncom` module exposes raw interfaces to Python. Many standard COM interfaces, such as ISTREAM and IDISPATCH, are exposed by equivalent Python objects, such as `PyIStream` and `PyIDispatch` objects. This module is rarely accessed directly. Instead, the win32com package provides classes and functions for additional services to a Python programmer.

The `win32com` package. The win32com package supports two subpackages:

- **`win32com.client.`** This package supports client-side COM—for example, using Python to access a spreadsheet in Microsoft Excel. The COM client helps Python to call COM interfaces.
- **`win32com.server.`** This package provides support for server-side COM—for example, creating a COM server in Python and using other languages, such as Visul Basic to implement COM interfaces to access Python objects. Therefore, this package can be used to create COM servers that can be accessed and manipulated by another COM client.

Creating COM Clients

To create COM clients, the methods and properties of the COM object have to be exposed using the IDISPATCH interface. To do this by using Python, you first need to import the `win32com.client` package and dispatch the right object by using the `win32com.client.dispatch()` method. This method takes the Prog ID or Class ID of the object you want to create as the parameter. For example, to open and manipulate a Microsoft Excel application, the Prog ID used should be `Excel.Applicaton`. This object also has a Class ID that uniquely registers it in the Windows registry. For example, to expose an Excel object by using Python, use the following statements:

```
>>> import win32com.client
>>> xl=win32com.client.Dispatch("Excel.Application")
```

Now, `xl` is the object representing Excel. To make it visible, type the following statement:

```
>>> xl.visible=1
```

By default, the win32com.client package uses late binding when creating an object. As stated earlier, late binding implies that Python does not have any advance knowledge of the properties and methods available for an object. For example, when Python encounters the statement,

```
xl.visible=1,
```

it first queries the object `xl` to determine whether the property `visible` exists. If so, the Python interpreter sets the property to 1.

The value in the object will be returned as:

```
>>> xl
<COMObject Excel.Application>
```

This implies that a COM object named `Excel.Application` exists.

The `win32com` package can also use early binding for COM objects. Using early binding implies that the methods and properties of the object are exposed before the type information is supplied by the object. Python uses the MakePy utility to support early binding. Composed in Python, the MakePy utility uses a COM type library to generate its Python source code that supports the interface. After you execute the MakePy utility, early binding is supported automatically. To make the MakePy utility available, you need to execute the `makepy.py` file in the `win32com\client` folder. To execute `makepy.py`, start PythonWin. On the Tools menu, select the COM MakePy utility. A list similar to the one presented in Figure A.3 appears. The items in this list depend on the applications installed on your machine.

From Type Library, select Microsoft Excel 9.0 Object Library (1.3) or another entry that supports the version of Microsoft Excel on your machine. Press the Enter key. After the Python interpreter has executed `makepy.py`, the following line will appear:

```
Generating to C:\Program
Files\Python22\win32com\gen_py\00020813-0000-0000-
C000-000000000046x0x1x3.py
```

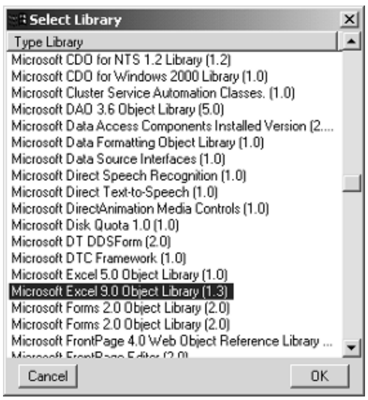


Figure A.3 List of object presented by the COM MakePy utility.

After creating the MakePy support, you can create the following code used earlier to create an `Excel.Application` object:

```
>>> import win32com.client
>>> xl=win32com.client.Dispatch("Excel.Application")
>>> xl.visible=1
```

When you print the `xl` object now, you obtain the following output:

```
>>> xl
<win32com.gen_py.Microsoft Excel 9.0 Object Library._Application>
```

Note that the Python interpreter has more information about the object as compared with the previous example that did not use MakePy. The Python interpreter now knows that the name of the type library is `Microsoft Excel 9.0 Object Library`.

After you create an Excel object, you can access its methods and set its properties. Let's examine a few statements to work with Excel using Python.

```
>>>xl.Workbooks.Add()
```

The preceding statement creates a new Excel workbook. To enter a value in cell A1 of the Excel worksheet, use the following statement:

```
>>> xl.Cells(1,1).Value="Hi!"
```

To set the time in cell A2, use the following statement:

```
>>> xl.Cells(1,2).Value="=NOW()"
```

To print the sum of 100 and 156 in cell B2, use the following statement:

```
>>>xl.Cells(2,2).Value="=SUM(100,156)"
```

You also use tuples to print multiple values in a cell range as follows:

```
>>> xl.Range("A3:D3").Value=('How','are','you!')
```

An Excel object can call a wide variety of functions and attributes. It is not possible to explain all of them because of the limited scope of the book; however, you can explore others by yourself.

Creating COM Servers

When you create a COM server, it implies exposing a Python object in a COM-aware environment, such as Visual Basic or Delphi. Implementing a COM object in Python requires implementing a Python class that exposes the functionality of the COM object. This class has two special attributes that determine how the object is published using COM:

`_reg_clsid_`. This attribute contains the class ID for the COM object. The Class ID can be generated using the `pythoncom.CreateGuid()` function.

`_reg_progid_`. This attribute contains a user-friendly string that you can use to call the COM object.

Another attribute of the implementing class is `_public_methods`. This attribute contains a list of the methods exposed by the class of the COM server. Therefore, the class that exports the functionality should implement the three attributes described in the following way:

```
class COMStringServer:
    _reg_clsid_='{BD055A03-EC10-4919-9F65-FDE57A840D1A}'
    _reg_progid_='COMSTRINGSERVER'
    _public_methods_=['letters','words']
```

Each COM object that you create should have a unique class ID. This ID is then used by the `_reg_clsid_` attribute. The class ID can be generated as follows:

```
>>> import pythoncom
>>> print pythoncom.CreateGuid()
{BD055A03-EC10-4919-9F65-FDE57A840D1A}
```

After creating the class, the COM object has to be registered. This can be done by using the following statements:

```
import win32com.server.register
win32com.server.register.UseCommandLine(COMStringServer)
```

When the `UseCommandLine()` function executes successfully, the Python interpreter displays a message containing the prog ID of the COM object. The message looks something like this:

```
Registered: COMSTRINGSERVER
```

Now, let's create a COM server that can be called by a COM client in Visual Basic. The server should calculate the number of letters and words in a string that is accepted from the user in a Visual Basic form. Therefore, the code for the COM server in Python will be:

```
class COMStringServer:
    _reg_clsid_='{BD055A03-EC10-4919-9F65-FDE57A840D1A}'
    _reg_progid_='COMSTRINGSERVER'
    _public_methods_=['letters','words']
    def letters(self,arg1):
        #arg1=arg1.strip()
        counter=arg1.count(' ')
        l=len(arg1)
        return l-counter
    def words(self,arg1):
        arg1=arg1.strip()
        counter=arg1.count(' ')
        return counter+1
if __name__=='__main__':
    import win32com.server.register
    win32com.server.register.UseCommandLine(COMStringServer)
```

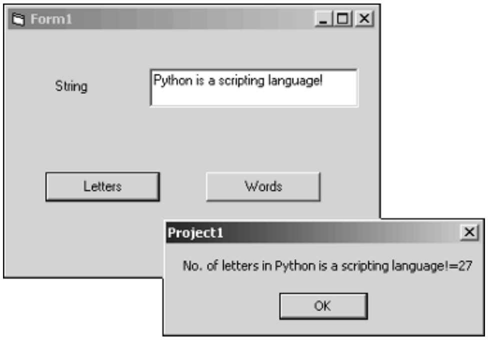


Figure A.4 A Visual Basic form design.

Save the preceding code as a `.py` file, and execute it by double-clicking on it in Windows explorer.

Create a Visual Basic COM client that can call the methods in the COM object. The Visual Basic form should look like the one in Figure A.4.

The COM server should be called at the time of loading the form. Therefore, the initialization steps should be written in the `Form_Load` function as follows:

```
Private Sub Form_Load()
Set COMStringServer = CreateObject("COMSTRINGSERVER")
End Sub
```

In the preceding statements, COMStringServer is the name of the object used by the VB client. This object can now be used to access the methods of the COM server. Therefore, the code for the VB client can be written as follows:

```
Dim COMStringServer As Object
Private Sub Command1_Click()
Dim output As Integer
output = COMStringServer.letters(Text1.Text)
MsgBox "No. of letters in " & Text1.Text & "=" & output
End Sub
Private Sub Command2_Click()
Dim output1 As Integer
output1 = COMStringServer.words(Text1.Text)
MsgBox "No. of words in " & Text1.Text & "=" & output1
End Sub
Private Sub Form_Load()
Text1 = ""
FuncCOMStringServer
End Sub
Private Sub Form_Unload(Cancel As Integer)
Set COMStringServer = Nothing
End Sub
Sub FuncCOMStringServer()
 Set COMStringServer = CreateObject("COMSTRINGSERVER")
End Sub
```



Figure A.5 Output shown when Letters button on the form is clicked.

When you execute the client, you will see an interactive window in which you can enter a string and determine the number of letters and words in it. If the input string is `Python is a scripting language1`, clicking the letters button will display a message box, as shown in Figure A.5.

Clicking the Words button will display a message box, as shown in Figure A.6.

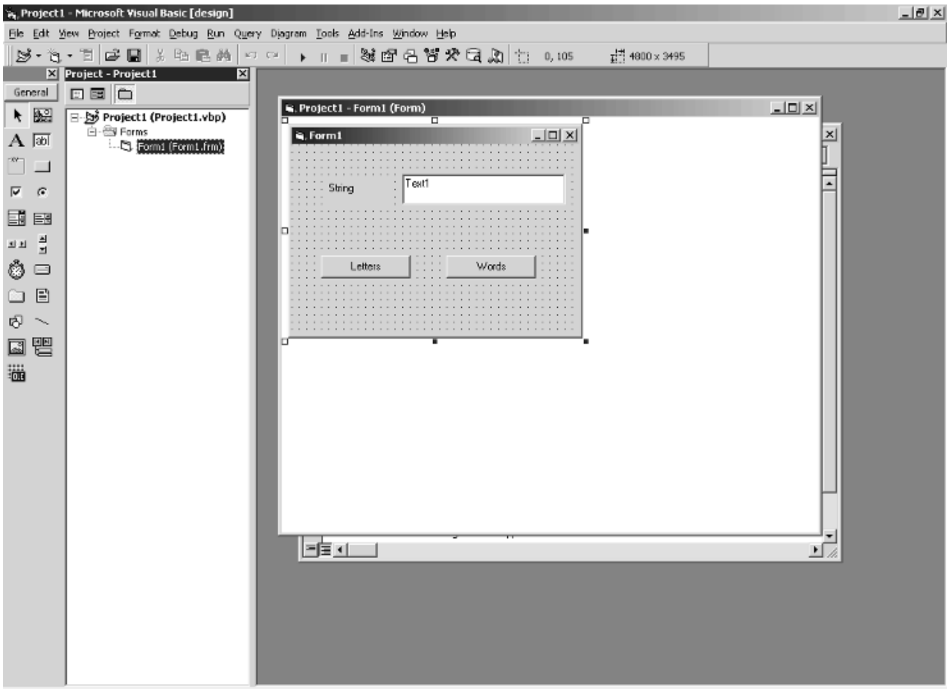


Figure A.6 Output shown when the Words button on the form is clicked.

Index

K

L

M

N

Q

R

S